Report No. 399

A CLASS OF ALGORITHMS FOR AUTOMATIC EVALUATION OF CERTAIN ELEMENTARY FUNCTIONS IN A BINARY COMPUTER

by

Bruce Gene De Lugish

June 1, 1970

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. 399

A CLASS OF ALGORITHMS FOR AUTOMATIC EVALUATION OF CERTAIN ELEMENTARY FUNCTIONS IN A BINARY COMPUTER*

by

Bruce Gene De Lugish

June 1, 1970

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

* This work was supported in part by the National Science Foundation under Grant No. US NSF GJ812 and Grant No. US NSF GJ813 and was submitted in partial fulfillment for the Doctor of Philosophy degree in Electrical Engineering, 1970.

ACKNOWLEDGMENT

The author wishes to express his sincerest gratitude to his advisor, Professor James E. Robertson. Professor Robertson's patience, guidance, invaluable aid and encouragement, timely suggestions, and continuing interest in this research are most appreciated.

The author also wishes to thank fellow student, Mr. Daniel E. Atkins, III, who often offered his time to discuss various aspects of the research and who proofread the entire text.

Further, the author wishes to thank Mrs. Norene McGhiey who typed the final version of the paper and Mr. Mark Goebel for drafting services.

Finally, the author wishes to express his gratitude to the National Science Foundation which provided funds for computer computations and to the Departments of Electrical Engineering and Computer Science which provided support in the form of assistantships and fellowships.

TABLE OF CONTENTS

Page

LIST OF FIGURES

A CLASS OF ALGORITHMS FOR AUTOMATIC EVALUATION OF CERTAIN ELEMENTARY FUNCTIONS IN A BINARY COMPUTER

Bruce Gene De Lugish
Department of Electrical Engineering
University of Illinois, 1970

The time required to evaluate elementary transcendental functions in a digital computer can often be significantly reduced by performing the algorithms in hardware rather than software form, providing that efficient hardware algorithms can be developed. Such time reduction may be important in batch processing on a large machine when the mix of problems includes frequent evaluation of such functions and may be essential in a special purpose real-time machine such as the guidance system of an airborne vehicle where an increase in speed is justified at almost any cost.

In situations which warrant the increase in hardware investment, primarily in control complexity, necessary to implement these algorithms, schemes which utilize redundant number recodings are also quite well justified. Ordinarily the introduction of redundancy, that is, allowing more than r digital values in radix r so that the representation of numbers is no longer unique, provides an increase in speed in exchange for some increase in hardware complexity. Usually the increase in hardware investment is moderate relative to the cost of the machine.

Algorithms to evaluate common elementary functions to register length precision in from one to three "multiplication cycle times" are developed; a "multiplication cycle time" is defined as the time required to perform, on the average, M low precision comparisons, M shifting operations, and M/3 additions where M is the length of the mantissa of a floating-point operand. The table below summarizes the results. A small register-speed read-only-memory containing less than 100 precomputed constants is required.

TABLE

Function	Multiplication Cycle Times
Division	1
Logarithm	1
Square Root	1
Exponential	3
Tangent	2
Cosine/Sine	2
Arctangent	1
Arccosine/ Arcsine	3

It may be possible to extend the techniques employed in the development of these algorithms to a much wider class of functions.

Algorithms which are readily amenable to implementation in radix $r = 2^n$, $n > 1$, promise even greater increases in speed than those limited to radix 2 implementation. Of the algorithms listed in the table, only the inverse trigonometric fail to fall in this category.

1. INTRODUCTION

1.1 Justification of the research effort

The continuing reduction of the size and cost and the increasing reliability of integrated circuit logic elements, especially LSI, encourages the utilization of more complex logic networks in the arithmetic unit of a digital computer. Furthermore, since in most large modern machines the cost of the arithmetic unit is only a small portion of the cost of the entire machine, it is possible to substantially increase the percentage cost of the arithmetic unit without appreciably affecting its relative cost. In arithmetic units these cost and size reductions encourage the replacement of very frequently used programmed subroutines by built-in function generators. Division is now almost universally hard-wired; algorithms for hard-wired square root have often been proposed. In future machines, especially those intended to perform substantial numbers of scientific calculations, it may well be feasible to hard-wire algorithms to evaluate logarithm, exponential, cosine/sine, tangent, and arctangent as well as square root. With these thoughts in mind, an investigation leading to the development of such algorithms was undertaken.

1.2 Redundancy and a measure of efficiency

The introduction of redundancy, for example allowing digital values $\bar{1}$, 0, 1 (where $\bar{1}$ means -1) in binary, allows, in general, several different representations of any particular number. Thus one can choose a recoding procedure which increases the probability of a zero, p_0, which increases the shift average $\langle S \rangle = 1/(1-p_0)$.

Making the usual assumption that 0 and 1 are equally likely, one can see that for conventionally represented numbers, $p_0 = 1/2$ and $\langle S \rangle = 2$.

It is known that by introducing minimal redundancy (three values $\bar{1}$, 0, 1 in radix 2) one can increase p_0 to 2/3 and $\langle S \rangle$ to 3. It is also known that one cannot achieve a higher shift average with minimal redundancy. The feasibility of further increasing redundancy in the algorithms developed in this research has not been studied because it is generally felt that the speed-hardware ratio diminishes rapidly.

1.3 Previous accomplishments

Very considerable research has been and is being directed toward formulation of fast division schemes, including, for example, the SRT[1] (and scaled SRT[2]) division algorithm which employs redundant representation of output digits to achieve greater speed. Several algorithms have also been developed for evaluation of various other elementary functions; included are the works of Meggitt,[3] Specker,[4] Senzig,[5] and Volder.[6] Most of the algorithms devised to date employ non-redundant number representation, although often digital values $\bar{1}$, 1 are used rather than 0, 1. The same factors which encourage hard-wiring of algorithms for evaluation of elementary functions also encourage the use of redundant number recodings which require more hardware in exchange for greater speed.

The analysis of the SRT division and a study of the correspondence between that division and multiplier recoding procedures carried out by Robertson,[7,8] Penhollow,[9] Frieman,[10] Shively,[11] and others lead to the conjecture by the author that redundancy techniques could be favorably employed in achieving greater efficiency in algorithms beyond division and square rooting.[12] That that conjecture holds some validity, at least under the assumptions discussed below, is verified in this paper.

1.4 Assumptions

It is assumed throughout this paper that the time required to perform addition is significantly greater than the time necessary to perform low precision comparisons, shifting operations, or complementations. It is further assumed that, with present hardware technology, one can economically build small (less than 100 words) read only memories which can be accessed at register speed. It is strongly felt that these assumptions are justified by presently available hardware.

1.5 Procedure

The fundamental technique employed throughout this research is the "normalization" of a given operand (or a simple function of that operand) by means of a continued product or continued summation, the terms of which may be easily chosen in a step-by-step process. The procedure for formulating an algorithm consists of the following operations: (1) choice of an appropriate function of a given operand to be "normalized"; (2) approximation of that function by a continued product or continued sum whose terms are easily chosen; (3) utilization of the individual terms comprising that product or sum, as they are chosen, to form the desired function; (4) choice of an appropriate rule for selecting the individual terms; (5) scaling of the operand upon which the selection rule is based so that the rule is the same for every step in the recursion. The procedure is best illustrated by the following example.

One scheme for performing division is to form the reciprocal of the divisor as a continued product of simple "multiplier" constants, using each of the "multipliers", as it is chosen, to form a better approximation to the quotient; viz.

$$q_{k+1} = q_k d_k, \; q_0 = \text{dividend}$$

where

$$\frac{1}{x} - \prod_{i=0}^{k} d_i \to 0.$$

In this particular scheme, the quantity

$$\frac{1}{x} - \prod_{i=0}^{k} d_i$$

is "normalized" to zero implicitly by explicitly forcing $x \prod_{i=0}^{k} d_i$ to unity. One must formulate a selection rule, based on the quantity $1 - x \prod_{i=0}^{k} d_i$, to choose the next multiplier. It is convenient to scale the partially normalized quantity, upon which the selection rule is based, in such a way that the selection rule is the same for every step of the recursion. A convenient scaling here is

$$u_k = 2^k(1 - x \prod_{i=0}^{k} d_i)$$

with the resulting recursion

$$u_{k+1} = 2u_k + s_k + 2^{-k} s_k u_k$$

where

$$d_k = 1 + s_k 2^{-k}$$

$$s_k = \begin{cases} \bar{1} \text{ if } u_k < -c \\ 0 \quad \text{otherwise} \\ 1 \text{ if } u_k \geq +c \end{cases}$$

for some comparison constant c. The efficiency of the algorithm is strongly dependent on the form of the recursion for two reasons: first, the more complicated the recursion the more time and/or hardware required to implement

it; second, the form of the recursion determines the difficulty of formulating a higher radix implementation which could lead to even greater speed.

This fundamental technique, termed the "normalization" technique in this paper, is discussed in detail in Section 2. In Sections 3 through 12 the technique is applied to various elementary functions--division, square rooting, logarithm, exponential, the circular functions, and their inverses. For each algorithm developed, an error bound and some methods of implementation are discussed. Finally, in Section 13, some considerations of importance in a higher radix implementation are discussed.

1.6 Results

With three adder circuit configurations of the type discussed in this paper, one can perform the algorithms listed below in the approximate average amounts of time indicated. A "multiplication cycle time" is the time required to perform M low precision (3 or 4 bits) comparisons, M shifting operations, and M/3 additions/subtractions, where M is the length of the mantissa.

TABLE 1

Function	Range of Operand Allowed	Multiplication Cycle Times
Division	Machine Range	1
Logarithm	$X > 0$	1
Square Root	$X \geq 0$	1
Exponential	Machine Range	3
Tangent	$0 \leq X < 2\pi$	2
Cosine/Sine	$0 \leq X < 2\pi$	2
Arctangent	Machine Range	1
Arccosine/Arcsine	$0 \leq X \leq 1$	3

Worst case error bounds are developed for each algorithm; typically worst case errors are less than one part in 2^{M+1}.

REFERENCES

1 J. E. Robertson, "A New Class of Digital Division Methods," IRE Transactions on Electronic Computers, EC-7:5:218-222, September, 1958.

2 G. A. Metze, "A Class of Binary Divisions Yielding Minimally Represented Quotients," IRE Transactions on Electronic Computers, EC-11:6:761-764, December, 1962.

3 J. E. Meggitt, "Pseudo Division and Pseudo Multiplication Processes," IBM Journal of Research & Development, 6:2:210-226, April, 1962.

4 W. H. Specker, "A Class of Algorithms for Ln X, Exp X, Sin X, Cos X, Tan^{-1} X, and Cot^{-1} X," IEEE Transactions on Electronic Computers, EC-14:1:85-86, February, 1965.

5 D. N. Senzig, "Digit-By-Digit Generation of the Trigonometric and Hyperbolic Functions," IBM Research Report, RC-860, December 17, 1962.

6 J. E. Volder, "The CORDIC Trigonometric Computing Technique," IEEE Transactions on Electronic Computers, EC-8:5:330-334, September, 1959.

7 J. E. Robertson, "Increasing the Efficiency of Digital Computer Arithmetic Through Use of Redundancy," Lecture Notes for EE 497B, University of Illinois, Fall Semester, 1964.

8 J. E. Robertson, "The Correspondence Between Methods of Digital Division and Multiplier Recoding Procedures," University of Illinois DCL Report No. 252, December 21, 1967.

9 J. O. Penhollow, "A Study of Arithmetic Recoding with Applications in Multiplication and Division," University of Illinois DCL Report No. 128, September 10, 1962.

10 C. V. Frieman, "Statistical Analysis of Certain Binary Division Algorithms," Proceedings of the IRE, 49:1:91-103, January, 1961.

11 R. R. Shively, "Stationary Distributions of Partial Remainders in SRT Digital Division," University of Illinois DCL Report No. 136, May 15, 1962.

12 G. A. Metze, "Minimal Square Rooting," IEEE Transactions on Electronic Computers, EC-14:2:181-185, April, 1965.

2. THE NORMALIZATION TECHNIQUE

2.1 Normalization schemes for division

2.1.1 Basic principle of normalization

Most of the division and square root algorithms known to the author appear to be derived on the basis of a hand-computation technique. Although some of the algorithms derived from the normalization technique used in this research have been known for some time, it is believed that the derivation by this technique provides further insight into the problem of automatic function evaluation because the normalization technique can be extended to algorithms beyond division and square root for which hand-computation methods are not known. Considerable attention is devoted in this paper to division and square root, not because the algorithms devised are better than those previously known (indeed, some coincide), but rather because it is believed that such a discussion leads to a fuller understanding of the general technique. With this thought in mind, let us begin with a normalization approach division algorithm.

Suppose one wishes to find the quotient of two numbers represented in normalized binary floating-point format,

$$X = x \cdot 2^{\alpha} \qquad\qquad x, y \in [1/2, 1)$$

$$Y = y \cdot 2^{\beta} \qquad\qquad \alpha, \beta \text{ integers.}$$

Let X be the divisor and Y be the dividend. The exponent of the quotient Q presents no particular problem, differing by at most one from the quantity $(\beta-\alpha)$.

$$Q = \frac{Y}{X} = \frac{y \cdot 2^{\beta}}{x \cdot 2^{\alpha}} = \frac{y}{x} \cdot 2^{(\beta-\alpha)} = q \cdot 2^{(\beta-\alpha)}$$

where $q = y/x$; $q \in (1/2, 2)$. One is thus able to concentrate his efforts on the fractional parts of the operands and devise a convenient algorithm for computing q.

Suppose one multiplies both the dividend and divisor by a sequence of (non-zero) constants $\{d_i\}$ such that the resulting divisor tends towards unity; one produces the quotient q.

$$q = \frac{y}{x} = \frac{y \prod_{i=0}^{M} d_i}{x \prod_{i=0}^{M} d_i} \cong y \prod_{i=0}^{M} d_i$$

$$\text{if} \quad x \prod_{i=0}^{M} d_i \cong 1.$$

Clearly the choice of the set of constants $\{d_i\}$ is critical, both for "normalizing" the divisor towards unity and for performing the indicated multiplications.

It is convenient to choose the set of constants $\{d_i\}$ to be of the form

$$d_i = 1 \pm 2^{-i}.$$

One might choose the positive sign if the partially normalized divisor is less than unity and the negative sign otherwise. Then each "multiplication" can be performed by a shift (of i places) and an addition (preceded by a complementation if the sign is negative). If two complete adder circuits are available, both the partially normalized divisor x_k and the partially developed quotient q_k are formed simultaneously and independently.

$$x_{k+1} = x_0 \prod_{i=0}^{k} d_i = x_k d_k, \quad x_0 = x$$

$$q_{k+1} = q_0 \prod_{i=0}^{k} d_i = q_k d_k, \quad q_0 = y.$$

The time required to perform a division with this algorithm is approximately the time required to do M additions, assuming that comparisons, shifts, and complements are very much faster than additions.

The above process is the normalization approach analog of the the non-restoring recoding in the following sense. Let us write

$$d_i = 1 + s_i 2^{-i}, \quad s_i = \{\bar{1}, 1\}.$$

Then the sequence $\{s_i\}$ represents a non-restoring recoding of the reciprocal of the divisor. Note, however, that the quotient is obtained in conventional form (digital values 0, 1) rather than in recoded form (digital values $\bar{1}$, 1).

Since in the above algorithm, the probability of choosing $s_i = 0$ is zero, the shift average $\langle S \rangle = 1/(1-p_0)$ is unity.

One can increase the shift average (and the speed) by allowing $s_i = 0$. For example, one might propose the following alternate selection rules: for the k^{th} step,

$$d_k = \begin{cases} 1 & \text{if } x_k > 1 - 2^{-k} \\ 1 + 2^{-k} & \text{if } x_k \leq 1 - 2^{-k}. \end{cases}$$

The division would then be completed in M/2 "addition cycle times," on the average, for a shift average of two. An "addition cycle time," in this context, includes the time required for comparison, shift, and complement,

as well as the addition itself. This alternate algorithm is the conventional recoding with digital values 0, 1.

Clearly, one can go a step further and introduce redundancy, that is, allow more than two digital values. In particular, one may allow digital values $\bar{1}$, 0, 1. If the algorithm is properly scaled (i.e., the proper selection rules are chosen), a shift average approaching three can be achieved. It is known[1] that a scale factor (essentially the ratio of comparison constant to multiplier) in the range [2/3, 5/6] yields a recoding for which the probability of a zero approaches its limit of 2/3. A convenient scale factor in the allowed range is 3/4.

Thus one may formulate the following algorithm for division: for the k^{th} step,

$$d_k = 1 + s_k 2^{-k}, \qquad 1 \leq k \leq M$$

$$s_k = \begin{cases} 1 & \text{if} \quad x_k < 1 - 3/4 \cdot 2^{-k} \\ 0 & \text{otherwise} \\ \bar{1} & \text{if} \quad x_k \geq 1 + 3/4 \cdot 2^{-k}. \end{cases}$$

(The initial multiplier d_0 may be chosen in a special way.) This division algorithm requires an average of only M/3 addition cycle times (if two adders are available) compared to M/2 addition cycle times for a conventional (0, 1) division and M addition cycle times for the common non-restoring ($\bar{1}$, 1) division. This division algorithm is discussed in detail in Section 3.

2.1.2 Other normalization division algorithms

While the division algorithm just proposed has been chosen for detailed study in this paper, three other normalization division schemes are known and are discussed here for completeness. To compare the four algorithms,

it is convenient to write each in a recursive form, as is done for the first algorithm here. Let

$$\alpha_{k+1} = 1 - x(\prod_{i=0}^{k} d_i)$$

$$= 1 - x_{k+1}$$

$$= 1 - x_k d_k$$

$$= 1 - (1 - \alpha_k)d_k$$

$$= 1 - d_k + \alpha_k d_k.$$

But $d_k = 1 + s_k 2^{-k}$, so

$$\alpha_{k+1} = -s_k 2^{-k} + \alpha_k(1+s_k 2^{-k})$$

$$\alpha_{k+1} = \alpha_k + s_k \alpha_k 2^{-k} - s_k 2^{-k} \tag{2-1}$$

where α_k approaches zero. This algorithm is a multiplicative reciprocal formation, i.e.,

$$\prod_{i=0}^{M} d_i \cong \frac{1}{x}$$

which simultaneously uses the digits of the reciprocal to form the quotient. Such a formulation of this algorithm leads quite naturally to an analogous additive reciprocal formation, represented by the following recursion.

$$\alpha_{k+1} = 1 - x(\sum_{i=0}^{k} d_i')$$

where $d_i' = s_i 2^{-i}$. Then,

$$\alpha_{k+1} = 1 - x_{k+1}$$

$$\alpha_{k+1} = 1 - x(\sum_{i=0}^{k} s_i 2^{-i})$$

$$= 1 - x(\sum_{i=0}^{k-1} s_i 2^{-i} + s_k 2^{-k})$$

$$= 1 - x(\sum_{i=0}^{k-1} s_i 2^{-i}) - x s_k 2^{-k}$$

$$= \alpha_k - s_k \, x \, 2^{-k} \qquad (2\text{-}2)$$

where α_k approaches zero, as before. This algorithm is nothing more than a bit by bit recoding of the reciprocal of the divisor,

$$\sum_{i=0}^{M} d_i' \cong \frac{1}{x} .$$

It offers the advantage of a simple recursion, but requires that the divisor be set aside in a special register throughout the division. Furthermore, and more importantly, as indicated by the appearance of the divisor in the recursion formula, in order to achieve a minimal recoding, the comparison constants in the selection rules must be scaled with respect to the divisor in a manner similar to that discussed below for the scaled square rooting algorithm. Since this algorithm offers no redeeming features, it is not proposed for implementation.

Both of the algorithms discussed above are reciprocal formation algorithms (although the quotient may be formed simultaneously). Slight alterations in these algorithms allow one to perform the division directly.

The following recursion is quite analogous to the algorithm represented by recursion relation (2-1).

$$\alpha_{k+1} = y - x \prod_{i=0}^{k} d_i$$

$$\alpha_{k+1} = y - x_{k+1}$$

$$= y - x_k\, d_k$$

$$= y - (y-\alpha_k)d_k$$

$$= y - y\, d_k + \alpha_k\, d_k$$

$$= y - y(1+s_k 2^{-k}) + \alpha_k(1+s_k 2^{-k})$$

$$= \alpha_k + \alpha_k\, s_k 2^{-k} - y\, s_k 2^{-k} \qquad (2\text{-}3)$$

where α_k approaches zero. Here

$$\prod_{i=0}^{M} d_i \cong \frac{y}{x} = q.$$

This algorithm has the curious property that the comparison constants must be scaled with respect to the dividend, rather than the divisor, to achieve a minimal recoding. One must again set aside an extra register, this time to hold the dividend. The only redeeming feature of this algorithm is that α_{M+1} is a remainder in the classical sense (the difference between the dividend and the product of the divisor and the quotient). This advantage is not sufficient to offset the necessity of scaling the comparison constants, and this algorithm is not proposed for implementation.

One may also propose a slight alteration in the algorithm represented by recursion relation (2-2).

$$\alpha_{k+1} = y - x(\sum_{i=0}^{k} d_i')$$

$$= y - x_{k+1}$$

$$\alpha_{k+1} = y - x\left(\sum_{i=0}^{k} s_i 2^{-i}\right)$$

$$= y - x\left(\sum_{i=0}^{k-1} s_i 2^{-i} + s_k 2^{-k}\right)$$

$$= y - x\left(\sum_{i=0}^{k-1} s_i 2^{-i}\right) - x\, s_k 2^{-k}$$

$$= \alpha_k - s_k\, x\, 2^{-k} \tag{2-4}$$

where α_k approaches zero, as before. This algorithm is a bit by bit recoding of the quotient,

$$\sum_{i=0}^{M} d_i' \cong \frac{y}{x} = q.$$

This last division algorithm, derived above by the normalization technique, may be recognized to be the well-known SRT division, which must be scaled with respect to the divisor to achieve a minimal recoding. Because this division has been analyzed so thoroughly elsewhere, it need not be discussed further here.

Thus two normalization approaches are known, one multiplicative (continued product) and one additive (continued summation). Each approach leads to a pair of possible division schemes. The basic requirement is simply to force either 1 - qx or y - qx to zero, forming q in any convenient manner. It is not known whether some combination of the above approaches, or perhaps an entirely new approach, would lead to a more efficient algorithm.

A similar situation is observed in square rooting, as discussed below. Many of the algorithms for other functions require a combination of multiplicative and additive techniques, and only one algorithm is known for

these functions. For the most basic function, multiplication, only the additive scheme leads to a feasible algorithm.

2.2 The standard (redundant) multiplication scheme

Suppose one wishes to form the product of two floating-point numbers X and Y. The exponent of the product P presents no problem, differing by at most one from the sum of the exponents of the given operands.

$$P = YX = y \cdot 2^{\beta} \cdot x \cdot 2^{\alpha} = yx \cdot 2^{(\beta+\alpha)} = p \cdot 2^{(\beta+\alpha)}$$

where $p = yx$, $p \in [1/4, 1)$. For multiplication, one proposes the following additive scheme, which is nothing other than the standard (redundant) multiplication algorithm.

$$p = yx = y(x - \sum_{i=0}^{M} m_i + \sum_{i=0}^{M} m_i)$$

where $m_i = s_i 2^{-i}$, $s_i = \{\bar{1}, 0, 1\}$. The selection rules are such that

$$x - \sum_{i=0}^{M} m_i \cong 0$$

so that

$$p \cong y \sum_{i=0}^{M} m_i .$$

Two simple recursions are performed simultaneously, but independently.

$$\begin{aligned} x_{k+1} &= x - \sum_{i=0}^{k} m_i \\ &= x - \sum_{i=0}^{k-1} m_i - m_k \\ &= x_k - m_k \end{aligned} \qquad (2\text{-}5)$$

$$p_{k+1} = y \sum_{i=0}^{k} m_i$$

$$= y \sum_{i=0}^{k-1} m_i + y\, m_k$$

$$= p_k + y\, m_k. \qquad (2\text{-}6)$$

Thus this multiplication algorithm can be performed in an average of M/3 addition cycle times. The details of this algorithm are discussed in Section 4.

It can be seen that the analogous multiplicative (continued product) formulation is not feasible. Here

$$p = yx = y \frac{x \prod_{i=0}^{M} m_i'}{\prod_{i=0}^{M} m_i'}$$

where $m_i' = 1 + s_i 2^{-i}$, $s_i = \{\bar{1}, 0, 1\}$. The selection rules would be such that

$$1 - x \prod_{i=0}^{M} m_i' \cong 0$$

so that

$$p = \frac{y}{\prod_{i=0}^{M} m_i'}.$$

Clearly it is not efficient to form p in such a manner. Whether some combination of the multiplicative and additive schemes might lead to an efficient algorithm is not known.

2.3 Normalization schemes for square rooting

As in the case of division, four basic normalization algorithms to perform square rooting are known: either the square root or its reciprocal can be formed; each can be formed by either a multiplicative or an additive scheme. Two of the algorithms must be scaled indirectly with respect to the root (directly with respect to the operand), one simply with respect to the operand, and one need not be scaled to achieve a minimal recoding.

Suppose one wishes to find the square root of X, $X \geq 0$. It is convenient, as a preparatory step, to normalize in a manner such that the exponent is even,

$$X = x \cdot 2^{\alpha'} \qquad x \in [1/4, 1)$$
$$\alpha' \text{ even integer.}$$

The exponent of the root is thus $\alpha'/2$ and is formed by shifting α'. One is thus able to concentrate his efforts on the fractional part of the operand and formulate a convenient algorithm for computing $r = \sqrt{x}$. Let us begin by considering a multiplicative approach that is quite analogous to the first division scheme proposed in Section 2.1.

One multiplies the given operand x by a sequence of (non-zero) constants $\{r_i\}$ such that the resulting operand tends towards unity.

$$x = \frac{x \prod_{i=0}^{M} r_i}{\prod_{i=0}^{M} r_i}$$

where

$$r_i = (1 + s_i 2^{-i})^2$$

$$r_i = 1 + s_i 2^{-(i-1)} + s_i^2\, 2^{-2i}, \qquad s_i = \{\bar{1}, 0, 1\}$$

$$1 - x \prod_{i=0}^{M} r_i \cong 0.$$

Thus,

$$\prod_{i=0}^{M} r_i = \prod_{i=0}^{M} (1 + s_i 2^{-i})^2 \cong \frac{1}{x}$$

$$\prod_{i=0}^{M} (1 + s_i 2^{-i}) \cong \frac{1}{\sqrt{x}}$$

so that,

$$y \prod_{i=0}^{M} (1 + s_i 2^{-i}) \cong \frac{y}{\sqrt{x}}$$

or

$$x \prod_{i=0}^{M} (1 + s_i 2^{-i}) \cong \sqrt{x}.$$

The selection rules for the choice of the set of multiplier constants $\{r_i\}$ are essentially the same as the selection rules for the division scheme represented by recursion formula (2-1). To carry out the algorithm, two recursions are performed simultaneously and independently.

$$\begin{aligned} \beta_{k+1} &= 1 - x(\prod_{i=0}^{k} r_i) \\ &= 1 - x_{k+1} \\ &= 1 - x_k r_k \\ &= 1 - (1 - \beta_k) r_k \end{aligned}$$

$$\beta_{k+1} = 1 - r_k + \beta_k r_k$$

But $r_k = 1 + s_k 2^{-(k-1)} + s_k^2 2^{-2k}$, so

$$\beta_{k+1} = \beta_k + (\beta_k - 1)(2s_k + s_k^2 2^{-k})2^{-k} \tag{2-7}$$

$$R_{k+1} = x \prod_{i=0}^{k} \sqrt{r_i}$$

$$= x \prod_{i=0}^{k} (1 + s_i 2^{-i})$$

$$= x \left[\prod_{i=0}^{k-1} (1 + s_i 2^{-i})\right](1 + s_k 2^{-k})$$

$$= R_k + s_k R_k 2^{-k} \tag{2-8}$$

where β_k approaches zero and R_k approaches the required root. Note that by initializing R_0 to an arbitrary value y, one can form the more general quantity $y/\sqrt{x}$ rather than the more common problem of finding $\sqrt{x}$ (when $y = x$). This algorithm is discussed in complete detail in Section 6.

The above algorithm is a multiplicative formation of the reciprocal of the square root, i.e.,

$$\prod_{i=0}^{M} \sqrt{r_i} \cong \frac{1}{\sqrt{x}}.$$

Such a multiplicative formulation leads one to investigate the feasibility of the analogous additive reciprocal root formation, represented by the following recursions.

$$\beta_{k+1} = 1 - x\left(\sum_{i=0}^{k} \sqrt{r_i'}\right)^2$$

$$= 1 - x_{k+1}$$

where $r_i' = (s_i 2^{-i})^2$. Then

$$\beta_{k+1} = 1 - x\left(\sum_{i=0}^{k-1} s_i 2^{-i} + s_k 2^{-k}\right)^2$$

$$= 1 - x\left[\left(\sum_{i=0}^{k-1} s_i 2^{-i}\right)^2 + 2s_k 2^{-k} \sum_{i=0}^{k-1} s_i 2^{-i} + s_k^2\, 2^{-2k}\right]$$

$$= 1 - x\left(\sum_{i=0}^{k-1} s_i 2^{-i}\right)^2 - 2s_k x\, 2^{-k} R_k + s_k^2\, 2^{-2k}$$

$$= \beta_k - 2s_k x\, R_k 2^{-k} + s_k^2\, 2^{-2k} \tag{2-9}$$

$$R_{k+1} = \sum_{i=0}^{k} \sqrt{r_i'}$$

$$= \sum_{i=0}^{k} s_i 2^{-i}$$

$$= \sum_{i=0}^{k-1} s_i 2^{-i} + s_k 2^{-k}$$

$$= R_k + s_k 2^{-k} \tag{2-10}$$

where β_k approaches zero and R_k approaches the reciprocal of the square root

of the given operand. From recursion (2-9) one can see that this algorithm cannot be performed efficiently because the indicated multiplication of x by R_k is indeed a full-scale multiplication. It is interesting to note also that the given operand must be set aside in a special register throughout the algorithm. Furthermore, in order to achieve a minimal recoding, the comparison constants must be scaled with respect to the root of the operand, (in practice one must convert this scaling to a scaling with respect to the operand itself, since, after all, the root is not known). This algorithm seems to offer no redeeming features and is not studied in detail or proposed for implementation.

Both of the square root algorithms discussed to this point are reciprocal root formation schemes, although in the first the root itself can easily be formed by appropriate initialization. The algorithms discussed below form the root directly.

An algorithm which is quite similar to that represented by recursion relations (2-7) and (2-8) but which forms the root directly is that represented by the following recursion relations.

$$\beta_{k+1} = x - \left(\prod_{i=0}^{k} r_i\right)$$

$$= x - \left(\prod_{i=0}^{k-1} r_i\right) r_k$$

$$= x - (x - \beta_k) r_k$$

$$= x - x\, r_k + \beta_k r_k$$

But $r_k = 1 + s_k 2^{-(k-1)} + s_k^2\, 2^{-2k}$, so

$$\beta_{k+1} = \beta_k + \beta_k(s_k 2^{-(k-1)} + s_k^2\, 2^{-2k}) - x(s_k 2^{-(k-1)} + s_k^2\, 2^{-2k})$$

$$= \beta_k + (\beta_k - x)(2s_k + s_k^2\, 2^{-k})2^{-k} \qquad (2\text{-}11)$$

$$R_{k+1} = \prod_{i=0}^{k} \sqrt{r_i}$$

$$= \prod_{i=0}^{k} (1 + s_i 2^{-i})$$

$$= \left[\prod_{i=0}^{k-1} (1 + s_i 2^{-i})\right](1 + s_k 2^{-k})$$

$$= R_k + s_k R_k 2^{-k} \qquad (2\text{-}12)$$

where β_k approaches zero and R_k approaches the root of the given operand. From relation (2-11) one can see that the comparison constants must be scaled with respect to the operand. Since this algorithm is less efficient than that first discussed, it is not proposed for implementation.

Finally, one may consider an additive algorithm which forms the root directly.

$$\beta_{k+1} = x - (\sum_{i=0}^{k} \sqrt{r_i'})^2$$

$$= x - (\sum_{i=0}^{k-1} \sqrt{r_i'} + \sqrt{r_k'})^2$$

$$= x - \{(\sum_{i=0}^{k-1} \sqrt{r_i'})^2 + 2\sqrt{r_k'} \sum_{i=0}^{k-1} \sqrt{r_i'} + r_k'\}$$

$$\beta_{k+1} = x - (\sum_{i=0}^{k-1} \sqrt{r'_i})^2 - 2\, s_k 2^{-k} R_k - s_k^2\, 2^{-2k}$$

$$= \beta_k - s_k R_k 2^{-(k-1)} - s_k^2\, 2^{-2k} \tag{2-13}$$

$$R_{k+1} = \sum_{i=0}^{k} \sqrt{r'_i}$$

$$= \sum_{i=0}^{k} s_i 2^{-i}$$

$$= R_k + s_k 2^{-k} \tag{2-14}$$

It is shown in Section 7 that this algorithm must be scaled indirectly with respect to the root of the given operand (in practice, directly with respect to the operand itself), but the recursions are so simple and convenient that it is feasible to propose an implementation. This algorithm is discussed fully in Section 7.

Hence, as in division, four normalization schemes are known; possibly others exist. Two algorithms from this set are studied in detail and are proposed for possible implementation because there is no clear case to be made for one in preference to the other--one is preferable in the strictly binary case, the other more readily amenable to a higher radix implementation.

2.4 Remarks on scaling

With regard to the necessity for scaling, the following table summarizes the observed requirements.

TABLE 2

(Requirements for Scaling)

	yx	$\frac{1}{x}$	$\frac{y}{x}$	$\frac{1}{\sqrt{x}}$	$\sqrt{x}$
Multiplicative (Continued Product)	--	None	wrt y	None	wrt x
Additive (Continued Sum)	None	None	wrt x	wrt $\sqrt{x}$	wrt $\sqrt{x}$

The necessity for scaling of the comparison constants in order to achieve a minimal recoding is basically an observed phenomenon which is not fully understood at this time. It is certainly an interesting question for further study.

2.5 Concluding remark

Most of the remainder of this paper is concerned with the derivation of those algorithms proposed for implementation, a discussion of convenient initialization, a brief study of implementation and hardware requirements, and a discussion of error bounds for the algorithms.

REFERENCE

1 J. E. Robertson, "Increasing the Efficiency of Digital Computer Arithmetic Through Use of Redundancy," Lecture Notes for EE 497B, University of Illinois, Fall Semester, 1964.

3. THE ALGORITHM FOR DIVISION

3.1 Basic algorithm

In Section 2.1.1, a division scheme is proposed which is discussed here in some detail.

It is assumed that one is attempting to formulate an algorithm for computing the quotient of two numbers represented in the usual binary format,

$$X = x \cdot 2^{\alpha} \qquad\qquad x, y \in [1/2, 1)$$

$$Y = y \cdot 2^{\beta} \qquad\qquad \alpha, \beta \text{ integers}$$

where X is the divisor and Y is the dividend. The quotient, as well as the divisor and dividend, is assumed to lie within machine range. As mentioned earlier, the exponent of the quctient presents no problem and one may concentrate on the ratio of fractional parts,

$$q = \frac{y}{x} .$$

Let us multiply both divisor and dividend by a sequence of (non-zero) constants $\{d_i\}$ chosen in such a way that the resulting divisor tends towards unity.

$$q = \frac{y}{x} = \frac{y \prod_{i=0}^{M} d_i}{x \prod_{i=0}^{M} d_i} \cong y \prod_{i=0}^{M} d_i$$

$$\text{if} \quad x \prod_{i=0}^{M} d_i \cong 1.$$

With the hardware configuration of Figure 1 (Section 3.5), both the partially normalized divisor x_k and the partially developed quotient q_k may be formed simultaneously and independently.

$$x_{k+1} = x_0 \prod_{i=0}^{k} d_i = x_k d_k, \qquad x_0 = x,$$

$$q_{k+1} = q_0 \prod_{i=0}^{k} d_i = q_k d_k, \qquad q_0 = y,$$

where $d_k = 1 + s_k 2^{-k}$, $1 \leq k \leq M$,

$$s_k = \begin{cases} 1 & \text{if} \quad x_k < 1 - c \cdot 2^{-k} \\ 0 & \qquad \text{otherwise} \\ \bar{1} & \text{if} \quad x_k \geq 1 + c \cdot 2^{-k}. \end{cases}$$

The initial multiplier d_0 is chosen in a special way. [See Section 3.2.]

It is known that a minimal recoding ($p_0 \cong 2/3$) will result if the comparison constant lies in the range [2/3, 5/6], the most convenient choice for c in this range being 3/4. However, it is also convenient to bound the variable

$$u_k = 2^k(x_k - 1),$$

upon which the actual comparisons are made, to lie in the range (-1, +1), so that each u_k can be represented in the machine with only a sign bit to the left of the radix point. This is easily accomplished by scaling down both the comparison constants and the multipliers by a factor of two.

Let

$$d_k = 1 + 1/2\ s_k 2^{-k} = 1 + s_k 2^{-(k+1)}, \quad 1 \leq k \leq M,$$

$$s_k = \begin{cases} 1 & \text{if} \quad x_k < 1 - 3/8 \cdot 2^{-k} \\ 0 & \qquad \text{otherwise} \\ \bar{1} & \text{if} \quad x_k \geq 1 + 3/8 \cdot 2^{-k} \end{cases}$$

or equivalently,

$$s_k = \begin{cases} 1 & \text{if} \quad u_k < -3/8 \\ 0 & \text{otherwise} \\ \overline{1} & \text{if} \quad u_k \geq +3/8 \end{cases}$$

where

$$u_k = 2^k(x_k - 1)$$

as previously indicated. It is shown that $|u_k| < 1$ for $k = 0, 1, \ldots, M$, within the desired range of $(-1, +1)$.

3.2 Choice of initialization step

The initial operand $x_0 = x$ lies in the range $[1/2, 1)$. The object of the normalization process is to force $x_{k+1} = x_0 \prod_{i=0}^{k} d_i$ to unity. The selection rules for the set of multipliers $\{d_i\}$ are essentially symmetric about unity, the only lack of symmetry lying in the choice of the placement of equality signs, so it would be convenient to choose the initial multiplier d_0 so as to force $x_1 = x_0 d_0$ to lie in a range symmetric about unity. Thus the following choice of initial multiplier would be desirable.

$$d_0 = \begin{cases} 2 \text{ (shift)} & \text{if} \quad 1/2 \leq x_0 < 2/3 \\ 1 & \text{if} \quad 2/3 \leq x_0 < 1. \end{cases}$$

This selection would leave x_1 in the range $(2/3, 4/3)$, symmetric about unity, and would make the choice of $s_k = \overline{1}$ and $s_k = 1$ later in the algorithm equally likely. However, the comparison constant of $2/3$ specified in this rule requires a full scale comparison in order to choose d_0. A less convenient, but much more practical choice of initialization step is the following.

$$d_0 = \begin{cases} 2 \text{ (shift)} & \text{if} \quad 1/2 \le x_0 < 3/4 \\ 1 & \text{if} \quad 3/4 \le x_0 < 1. \end{cases}$$

Then,

$$x_1 \in [3/4,\ 3/2)$$

and the probabilities, for a finite register length (where the skewness in the probability density of x_k has not yet dissipated), that $s_k = \bar{1}$ and $s_k = 1$ are not equal. However, as verified by experimental means, the probability that $s_k = 0$ is still very nearly 2/3, and that is the critical factor. Refer to Section 3.4.

Having chosen an initialization step, the algorithm is now completely specified. An example illustrates the flow of the algorithm.

<u>Example</u>: Numerical values of x_k and q_k are listed below, rounded to 14 decimal places. The divisor $x = 0.6_{10} = 0.1001\overline{1001}_2$ (where the overbar indicates a repeated digit pattern) and the divisor $y = 0.5_{10} = 0.1000\overline{0000}$. The correct quotient, also rounded to 14 decimal places, is $q = 0.83333333333333$.

For this divisor, $d_0 = 2$, so that $x_1 = 1.2$ and $q_1 = 1.0$.

TABLE 3

k	s_k	x_{k+1}	q_{k+1}
1	0	1.20000000000000	1.00000000000000
2	-1	0.90000000000000	0.75000000000000
3	1	1.01250000000000	0.84375000000000
4	0	1.01250000000000	0.84375000000000
5	0	1.01250000000000	0.84375000000000
6	-1	0.99667968750000	0.83056640625000
7	0	0.99667968750000	0.83056640625000
8	1	1.00057296752930	0.83381080627441
9	0	1.00057296752930	0.83381080627441
10	0	1.00057296752930	0.83381080627441
11	-1	1.00008440651000	0.83340367209166
12	0	1.00008440651000	0.83340367209166
13	0	1.00008440651000	0.83340367209166
14	-1	1.00002336620198	0.83335280516832
15	-1	0.99999284791078	0.83332737325898
16	0	0.99999284791078	0.83332737325898
17	1	1.00000047725074	0.83333373104228
18	0	1.00000047725074	0.83333373104228
19	0	1.00000047725074	0.83333373104228
20	0	1.00000047725074	0.83333373104228
21	-1	1.00000000004136	0.83333333367780
22	0	1.00000000004136	0.83333333367780
23	0	1.00000000004136	0.83333333367780
24	0	1.00000000004136	0.83333333367780
25	0	1.00000000004136	0.83333333367780

(Continued)

TABLE 3 (Continued)

k	s_k	x_{k+1}	q_{k+1}
26	0	1.00000000004136	0.83333333367780
27	0	1.00000000004136	0.83333333367780
28	0	1.00000000004136	0.83333333367780
29	0	1.00000000004136	0.83333333367780
30	0	1.00000000004136	0.83333333367780
31	-1	0.99999999994769	0.83333333328975
32	0	0.99999999994769	0.83333333328975
33	0	0.99999999994769	0.83333333328975
34	1	1.00000000000590	0.83333333333825
35	0	1.00000000000590	0.83333333333825
36	0	1.00000000000590	0.83333333333825
37	-1	0.99999999999863	0.83333333333219
38	0	0.99999999999863	0.83333333333219
39	1	1.00000000000044	0.83333333333370
40	0	1.00000000000044	0.83333333333370

Thus the quotient produced in 40 steps of this algorithm is

$$q_{M+1} = 0.83333333333370$$

which differs from the correct quotient by 0.37×10^{-12}. The error bound, derived in the next section, is 0.45×10^{-12} for $M = 40$ steps.

3.3 Error bound

Given an initial operand $x_0 = x$ in the range [1/2, 1) and the selection rules listed in the last section for the choice of the set of multipliers, one may now produce a bound on x_{M+1}, the resulting divisor, and ultimately an error bound for the division algorithm itself. Since,

$$d_0 = \begin{cases} 2 & \text{if} \quad 1/2 \leq x_0 < 3/4 \\ 1 & \text{if} \quad 3/4 \leq x_0 < 1 \end{cases}$$

then

$$x_1 = x_0 d_0 \in [3/4, 3/2)$$

and

$$u_1 = 2(x_1 - 1) \in [-1/2, 1).$$

Next let us find the range of $x_2 = x_1 d_1$. The selection rule for the first ($k = 1$) step is,

$$d_1 = 1 + 1/4\ s_1$$

$$s_1 = \begin{cases} 1 & \text{if} \quad 3/4 \leq x_1 < 13/16 \\ 0 & \text{if} \quad 13/16 \leq x_1 < 19/16 \\ \bar{1} & \text{if} \quad 19/16 \leq x_1 < 3/2. \end{cases}$$

The first range, [3/4, 13/16) maps onto 5/4 [3/4, 13/16) or [15/16, 65/64); the second range, [13/16, 19/16) maps onto itself; the third range, [19/16,3/2), maps onto 3/4 [19/16, 3/2) or [57/64, 9/8). Hence,

$$x_2 \in [13/16, 19/16),$$

so that x_2 lies in the middle range of the selection rules for s_1. It is proved by induction that similar behavior occurs later in the process.

Hypothesis: For $k \geq 2$, $x_k \in [1 - 3/8 \cdot 2^{-(k-1)},\ 1 + 3/8 \cdot 2^{-(k-1)})$.

The hypothesis has been shown explicitly to be valid for $k = 2$.

Proof: For some $k \geq 2$, $x_k \in [1 - 3/8 \cdot 2^{-(k-1)},\ 1 + 3/8 \cdot 2^{-(k-1)})$ or, equivalently, $x_k \in [1 - 3/4 \cdot 2^{-k},\ 1 + 3/4 \cdot 2^{-k})$. For all $k \geq 2$,

$$d_k = 1 + 1/2\, s_k 2^{-k}$$

$$s_k = \begin{cases} 1 & \text{if} \quad x_k < 1 - 3/8 \cdot 2^{-k} \\ 0 & \text{otherwise} \\ \bar{1} & \text{if} \quad x_k \geq 1 + 3/8 \cdot 2^{-k}. \end{cases}$$

The first range,

$$[1 - 3/4 \cdot 2^{-k},\ 1 - 3/8 \cdot 2^{-k})$$

maps onto

$$(1 + 1/2 \cdot 2^{-k}) \cdot [1 - 3/4 \cdot 2^{-k},\ 1 - 3/8 \cdot 2^{-k}),$$

or

$$[1 - 1/4 \cdot 2^{-k} - 3/8 \cdot 2^{-2k},\ 1 + 1/8 \cdot 2^{-k} - 3/16 \cdot 2^{-2k}),$$

which lies within the desired range of

$$[1 - 3/8 \cdot 2^{-k},\ 1 + 3/8 \cdot 2^{-k}).$$

The second range,

$$[1 - 3/8 \cdot 2^{-k},\ 1 + 3/8 \cdot 2^{-k})$$

maps onto itself, and thus lies within the desired range.

The third range,

$$[1 + 3/8 \cdot 2^{-k},\ 1 + 3/4 \cdot 2^{-k})$$

maps onto

$$(1 - 1/2 \cdot 2^{-k}) \quad [1 + 3/8 \cdot 2^{-k}, \ 1 + 3/4 \cdot 2^{-k})$$

or

$$[1 - 1/8 \cdot 2^{-k} - 3/16 \cdot 2^{-2k}, \ 1 + 1/4 \cdot 2^{-k} - 3/8 \cdot 2^{-2k}),$$

which again lies within the desired range. Hence,

$$x_{k+1} \in [1 - 3/8 \cdot 2^{-k}, \ 1 + 3/8 \cdot 2^{-k})$$

for all $k \geq 2$. Furthermore, $|u_k| \leq 3/4$ for all $k \geq 2$ and $|u_k| < 1$ for all k.

Q.E.D.

Therefore, the final divisor

$$x_{M+1} \in [1 - 3/8 \cdot 2^{-M}, \ 1 + 3/8 \cdot 2^{-M})$$

so that the error in the final divisor is bounded by

$$|x_{M+1} - 1| \leq 3/8 \cdot 2^{-M}.$$

The algorithm is thus capable of producing M correct quotient bits in M steps beyond the initialization, the error in the algorithm being less than $2^{-(M+1)}$, neglecting round-off.

3.4 Experimental estimate of speed

The theoretical prediction that, with the selection rules chosen, the probability that s_k be zero is 2/3 is an asymptotic result, that is, it assumes a register of infinite length. To test the value of the algorithm in any practical machine, it becomes necessary to estimate the probability of a zero for a relatively short register. For this reason, this division algorithm was simulated on the available IBM 360/75 computer system for a register length M of 40 bits, and a Monte Carlo estimate of the probability

of a zero was made. For a statistically significant sample of 2^{18} pseudo-random divisor operands, approximately uniform over the range [1/2, 1), the mean probability of a zero is 0.676, with a corresponding shift average of 3.08. As a practical matter, these figures differ very little from their theoretically predicted asymptotic values of 2/3 and 3.00. The relevant statistical considerations are discussed in Appendix A.

3.5 Implementation

The necessary recursion formulas to implement the division process are those given earlier, namely,

$$x_{k+1} = x_0 \prod_{i=0}^{k} d_i = x_k d_k, \quad x_0 = x,$$

$$q_{k+1} = q_0 \prod_{i=0}^{k} d_i = q_k d_k, \quad q_0 = y,$$

except that the former recursion must be rewritten in terms of $u_k = 2^k(x_k - 1)$ so that the comparison constants remain fixed at $\pm$ 3/8 (except for the initialization).

$$u_{k+1} = 2u_k + s_k + s_k u_k 2^{-k} \qquad (3\text{-}1)$$

$$q_{k+1} = q_k + 2^{-(k+1)} s_k q_k \qquad (3\text{-}2)$$

where

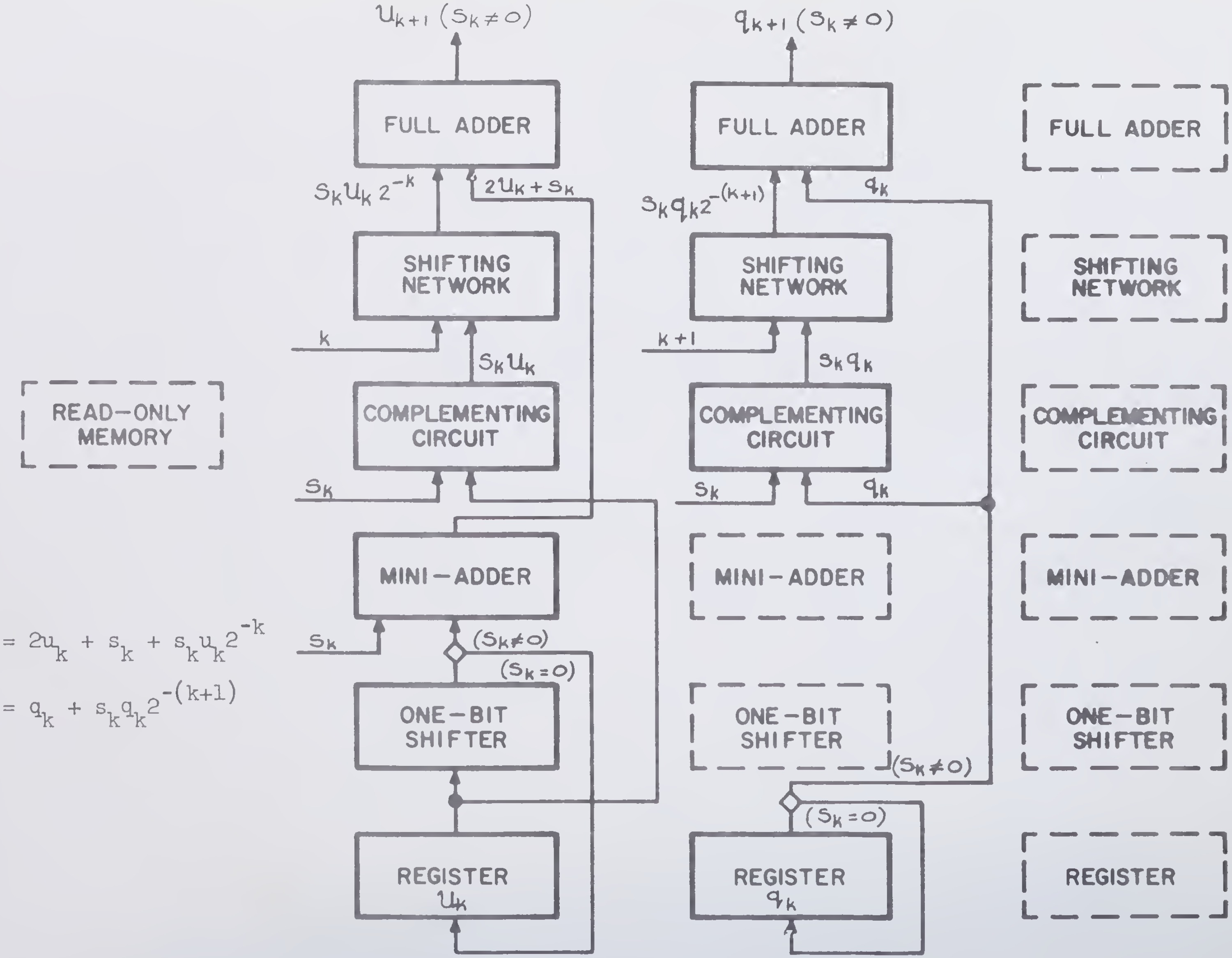
$$s_k = \begin{cases} 1 & \text{if} \quad u_k < -3/8 \\ 0 & \quad\quad \text{otherwise} \\ \overline{1} & \text{if} \quad u_k \geq +3/8. \end{cases}$$

To implement the first of these recursion relations, one requires a counter (0 to M) to keep track of the step being performed, a comparison circuit to select the value of s_k (represented by a sign bit and a magnitude bit), a register to hold x_k (eventually x_{k+1}), a complementing circuit to form the negative of a given operand, a shifting network capable of rapidly shifting k places ($0 \leq k < M$), a full adder (which probably includes the register mentioned above), and facilities for shifting a single bit position. To implement the second recursion, one requires an additional register to hold q_k (probably falling within the full adder again), another complementing circuit and shifting network, and another full adder as indicated. Of course, control circuitry is also required, but that may be provided in either hardware or microprogramming form. The control requirement is a design criterion which is highly dependent on other machine design parameters and cannot be discussed in this paper in any detail.

A hardware structure sufficient to implement all algorithms discussed in this paper has been developed in block diagram form. The flow of information for the division algorithm is indicated in Figure 1. Those boxes which are not required by the division algorithm are shown in dashed form; the counter and comparison circuitry are not explicitly shown.

The read-only memory indicated in Figure 1 is required by several functions in the set; its size is a function of the register length, as well as which functions in the set are to be implemented in the machine. To implement all of the functions discussed, a total of slightly more than 4M/3 words need be stored in the memory.

The "mini-adder" is an adder capable of adding only a single bit (in any digital position) to a full precision operand. The "one-bit shifter" is capable of shifting left one digital position.



$$u_{k+1} = 2u_k + s_k + s_k u_k 2^{-k}$$

$$q_{k+1} = q_k + s_k q_k 2^{-(k+1)}$$

FIGURE 1. Block diagram for division

3.6 Concluding remark

A division algorithm using an implicit redundant recoding and yielding a shift average of approximately three has been proposed. It has been shown that the algorithm can be realized with reasonable cost in hardware, especially with the expected advances in hardware technology (LSI). It is shown later in this paper that the structure of Figure 1 is sufficient to implement the other elementary functions in the set as well.

While considerable discussion has been devoted to division techniques, it should be kept clearly in mind that the avowed purpose of this research was not to develop new division schemes, although that has been done, but rather to extend the techniques used in those schemes to the elementary transcendental functions.

4. THE ALGORITHM FOR MULTIPLICATION

4.1 Basic algorithm

In Section 2.2, it is shown that an additive normalization scheme leads to a feasible algorithm for multiplication, but that a multiplicative normalization scheme does not. The details of the additive scheme are discussed here.

In formulating an algorithm for multiplication, one need only be concerned with the product of the fractional parts y and x of the multiplicand Y and the multiplier X.

$$p = yx = y\left(x - \sum_{i=0}^{M} m_i + \sum_{i=0}^{M} m_i\right)$$

where

$$m_k = 1/2\ s_k 2^{-k}, \quad s_k = \{\bar{1}, 0, 1\}.$$

The selection rules must be chosen in such a way that

$$x - \sum_{i=0}^{M} m_i \cong 0$$

so that

$$p \cong y \sum_{i=0}^{M} m_i .$$

The selection rule at the k^{th} step for the selection of s_k is virtually the same as that for division, except that the operand is being forced to zero rather than unity.

$$s_k = \begin{cases} \bar{1} & \text{if} \quad x_k < -3/8 \cdot 2^{-k} \\ 0 & \quad\quad \text{otherwise} \\ 1 & \text{if} \quad x_k \geq +3/8 \cdot 2^{-k} \end{cases}$$

where

$$x_{k+1} = x_0 - \sum_{i=0}^{k} m_i$$

$$= x_0 - \sum_{i=0}^{k-1} m_i - m_k$$

$$= x_k - m_k, \qquad x_0 = x$$

$$p_{k+1} = y \sum_{i=0}^{k} m_i$$

$$= y \sum_{i=0}^{k-1} m_i + ym_k$$

$$= p_k + ym_k, \qquad p_0 = 0.$$

Letting $u_k = 2^k x_k$, one may write the two recursions which must be implemented to perform multiplication.

$$u_{k+1} = 2u_k - s_k \tag{4-1}$$

$$p_{k+1} = p_k + ys_k 2^{-(k+1)} \tag{4-2}$$

where the selection rule for s_k now reads,

$$s_k = \begin{cases} \bar{1} & \text{if} \quad u_k < -3/8 \\ 0 & \quad \text{otherwise} \\ 1 & \text{if} \quad u_k \geq +3/8. \end{cases}$$

Note that a register must be provided to hold the original multiplicand throughout the process.

4.2 Choice of initialization step

The initial operand $x_0 = x$ lies in the range $[1/2, 1)$. The object of the normalization process is to force $x_{k+1} = x_0 - \sum_{i=0}^{k} m_i$ to zero. The selection rules for the set of constants $\{m_i\}$ are essentially symmetric about zero, and it is convenient to choose the initial multiplier m_0 so as to force $x_1 = x_0 - m_0$ to lie in a range symmetric about zero, the extent of the range being as small as possible. Further, m_0 should contain no more than a single non-zero bit since one would like to form

$$p_1 = p_0 + ym_0$$

in no more than one addition cycle time. The choice of

$$m_0 = \begin{cases} 1/2 & \text{if} \quad 1/2 \leq x_0 < 3/4 \\ 1 & \text{if} \quad 3/4 \leq x_0 < 1 \end{cases}$$

satisfies all of these criteria since it contains only a single non-zero bit and leaves x_1 in the range $[-1/4, +1/4)$. It is shown in the next section that such a choice of initialization leads to a convergent algorithm.

An example illustrates the flow of the algorithm.

Example: With a multiplicand of 0.5 and a multiplier of 0.6, the correct product is 0.3.

For this multiplier, $m_0 = 1/2$, so that $x_1 = 0.1$ and $p_1 = 0.25$.

TABLE 4

k	s_k	x_{k+1}	p_{k+1}
1	0	0.10000000000000	0.25000000000000
2	1	-0.02500000000000	0.31250000000000
3	0	-0.02500000000000	0.31250000000000
4	-1	0.00625000000000	0.29687500000000
5	0	0.00625000000000	0.29687500000000
6	1	-0.00156250000000	0.30078125000000
7	0	-0.00156250000000	0.30078125000000
8	-1	0.00039062500000	0.29980468750000
9	0	0.00039062500000	0.29980468750000
10	1	-0.00009765625000	0.30004882812500
11	0	-0.00009765625000	0.30004882812500
12	-1	0.00002441406250	0.29998779296875
13	0	0.00002441406250	0.29998779296875
14	1	-0.00000610351563	0.30000305175781
15	0	-0.00000610351563	0.30000305175781
16	-1	0.00000152587891	0.29999923706055
17	0	0.00000152587891	0.29999923706055
18	1	-0.00000038146973	0.30000019073486
19	0	-0.00000038146973	0.30000019073486
20	-1	0.00000009536743	0.29999995231628
21	0	0.00000009536743	0.29999995231628
22	1	-0.00000002384188	0.30000001192093
23	0	-0.00000002384188	0.30000001192093
24	-1	0.00000000596046	0.29999999701977
25	0	0.00000000596046	0.29999999701977

(Continued)

TABLE 4 (Continued)

k	s_k	x_{k+1}	p_{k+1}
26	1	-0.0000000149012	0.3000000074506
27	0	-0.0000000149012	0.3000000074506
28	-1	0.00000000037253	0.2999999981374
29	0	0.00000000037253	0.2999999981374
30	1	-0.0000000009313	0.3000000004657
31	0	-0.0000000009313	0.3000000004657
32	-1	0.0000000002328	0.2999999998836
33	0	0.0000000002328	0.2999999998836
34	1	-0.00000000000582	0.3000000000291
35	0	-0.00000000000582	0.3000000000291
36	-1	0.000000000000146	0.29999999999927
37	0	0.000000000000146	0.29999999999927
38	1	-0.00000000000036	0.30000000000018
39	0	-0.00000000000036	0.30000000000018
40	-1	0.000000000000009	0.29999999999995

Thus the product generated by the algorithm in 40 steps is

$$p_{M+1} = 0.29999999999995$$

which differs from the correct product by 0.5×10^{-13}. The error bound, derived in the next section, is 0.34×10^{-12} for $M = 40$.

It may be observed that the sequence of values of s_k takes on a peculiar pattern for this multiplier: $010\bar{1}\overline{010\bar{1}}$. To dispel any inference that such a pattern exists for all multipliers, let us change the multiplier slightly and repeat the example.

<u>Example</u>: With $y = 0.5$ and $x = 0.61$, $p = 0.305$. Also, $m_0 = 1/2$, $x_1 = 0.11$, $p_1 = 0.25$.

TABLE 5

k	s_k	x_{k+1}	p_{k+1}
1	0	0.11000000000000	0.25000000000000
2	1	-0.01500000000000	0.31250000000000
3	0	-0.01500000000000	0.31250000000000
4	0	-0.01500000000000	0.31250000000000
5	-1	0.00062500000000	0.30468750000000
6	0	0.00062500000000	0.30468750000000
7	0	0.00062500000000	0.30468750000000
8	0	0.00062500000000	0.30468750000000
9	0	0.00062500000000	0.30468750000000
10	1	0.00013671875000	0.30493164062500
11	0	0.00013671875000	0.30493164062500
12	1	0.00001464843750	0.30499267578125
13	0	0.00001464843750	0.30499267578125
14	0	0.00001464843750	0.30499267578125
15	1	-0.00000061035156	0.30500030517578
16	0	-0.00000061035156	0.30500030517578
17	0	-0.00000061035156	0.30500030517578
18	0	-0.00000061035156	0.30500030517578
19	0	-0.00000061035156	0.30500030517578
20	-1	-0.00000013351440	0.30500006675720
21	0	-0.00000013351440	0.30500006675720
22	-1	-0.00000001430511	0.30500000715256
23	0	-0.00000001430511	0.30500000715256
24	0	-0.00000001430511	0.30500000715256
25	-1	0.00000000059605	0.30499999970198

(Continued)

TABLE 5 (Continued)

k	s_k	x_{k+1}	p_{k+1}
26	0	0.00000000059605	0.30499999970198
27	0	0.00000000059605	0.30499999970198
28	0	0.00000000059605	0.30499999970198
29	0	0.00000000059605	0.30499999970198
30	1	0.00000000013039	0.30499999993481
31	1	0.00000000013039	0.30499999993481
32	1	0.00000000001397	0.34099999999302
33	0	0.00000000001397	0.34099999999302
34	0	0.00000000001397	0.34099999999302
35	1	-0.00000000000058	0.30500000000029
36	0	-0.00000000000058	0.30500000000029
37	0	-0.00000000000058	0.30500000000029
38	0	-0.00000000000058	0.30500000000029
39	0	-0.00000000000058	0.30500000000029
40	-1	-0.00000000000013	0.30500000000006

4.3 Error bound

Given an initial operand $x_0 = x$ in the range $[1/2, 1)$ and the selection rules listed in the first section for the choice of the set of constants, one may produce a bound on x_{M+1}, and ultimately an error bound for the multiplication algorithm.

It has already been noted that, with the previously indicated choice of m_0,

$$x_1 \in [-1/4, +1/4).$$

Also,

$$u_1 = 2x_1 \in [-1/2, +1/2).$$

Following the example of the division algorithm, one may construct an inductive error bound proof.

Hypothesis: For $k \geq 1$, $x_k \in [-3/8 \cdot 2^{-(k-1)}, \; +3/8 \cdot 2^{-(k-1)})$.

The hypothesis has been shown explicitly to be valid for $k = 1$.

Proof: For some $k \geq 1$, $x_k \in [-3/8 \cdot 2^{-(k-1)}, \; +3/8 \cdot 2^{-(k-1)})$ or $x_k \in [-3/4 \cdot 2^{-k}, \; +3/4 \cdot 2^{-k})$. For all $k \geq 1$,

$$m_k = 1/2 \, s_k 2^{-k}$$

$$s_k = \begin{cases} \bar{1} & \text{if} \quad x_k < -3/8 \cdot 2^{-k} \\ 0 & \quad\;\;\, \text{otherwise} \\ 1 & \text{if} \quad x_k \geq +3/8 \cdot 2^{-k}. \end{cases}$$

The first range,

$$[-3/4 \cdot 2^{-k}, \; -3/8 \cdot 2^{-k})$$

maps onto

$$[-3/4 \cdot 2^{-k}, \; -3/8 \cdot 2^{-k}) + 1/2 \cdot 2^{-k}$$

or

$$[-1/4 \cdot 2^{-k}, \; +1/8 \cdot 2^{-k}),$$

which lies within the desired range of

$$[-3/8 \cdot 2^{-k}, \; +3/8 \cdot 2^{-k}).$$

The second range,

$$[-3/8 \cdot 2^{-k}, \; +3/8 \cdot 2^{-k})$$

maps onto itself, and thus lies within the desired range.

The third range,

$$[+3/8 \cdot 2^{-k}, \ +3/4 \cdot 2^{-k})$$

maps onto

$$[+3/8 \cdot 2^{-k}, \ +3/4 \cdot 2^{-k}) - 1/2 \cdot 2^{-k}$$

or

$$[-1/8 \cdot 2^{-k}, \ +1/4 \cdot 2^{-k})$$

again within the desired range. Hence,

$$x_{k+1} \in [-3/8 \cdot 2^{-k}, \ +3/8 \cdot 2^{-k})$$

for all $k \geq 1$. Furthermore, $|u_k| \leq 3/4$ for all $k \geq 1$ and $|u_k| < 1$ for all k, since $u_0 \in [1/2, 1)$.

Q.E.D.

Therefore,

$$|x_{M+1}| \leq 3/8 \cdot 2^{-M}.$$

The error in the multiplication algorithm,

$$yx_{M+1}$$

is thus less, in magnitude, than $3/8 \cdot 2^{-M}$ and the algorithm is capable of producing M correct product bits in M steps beyond the initialization.

4.4 Experimental estimate of speed

The Monte Carlo estimate of the mean probability of a zero is 0.684, with a corresponding shift average of 3.17.

4.5 Implementation

The recursion formulas necessary to implement the multiplication algorithm are those given in (4-1) and (4-2), repeated here for reference purposes.

$$u_{k+1} = 2u_k - s_k \tag{4-1}$$

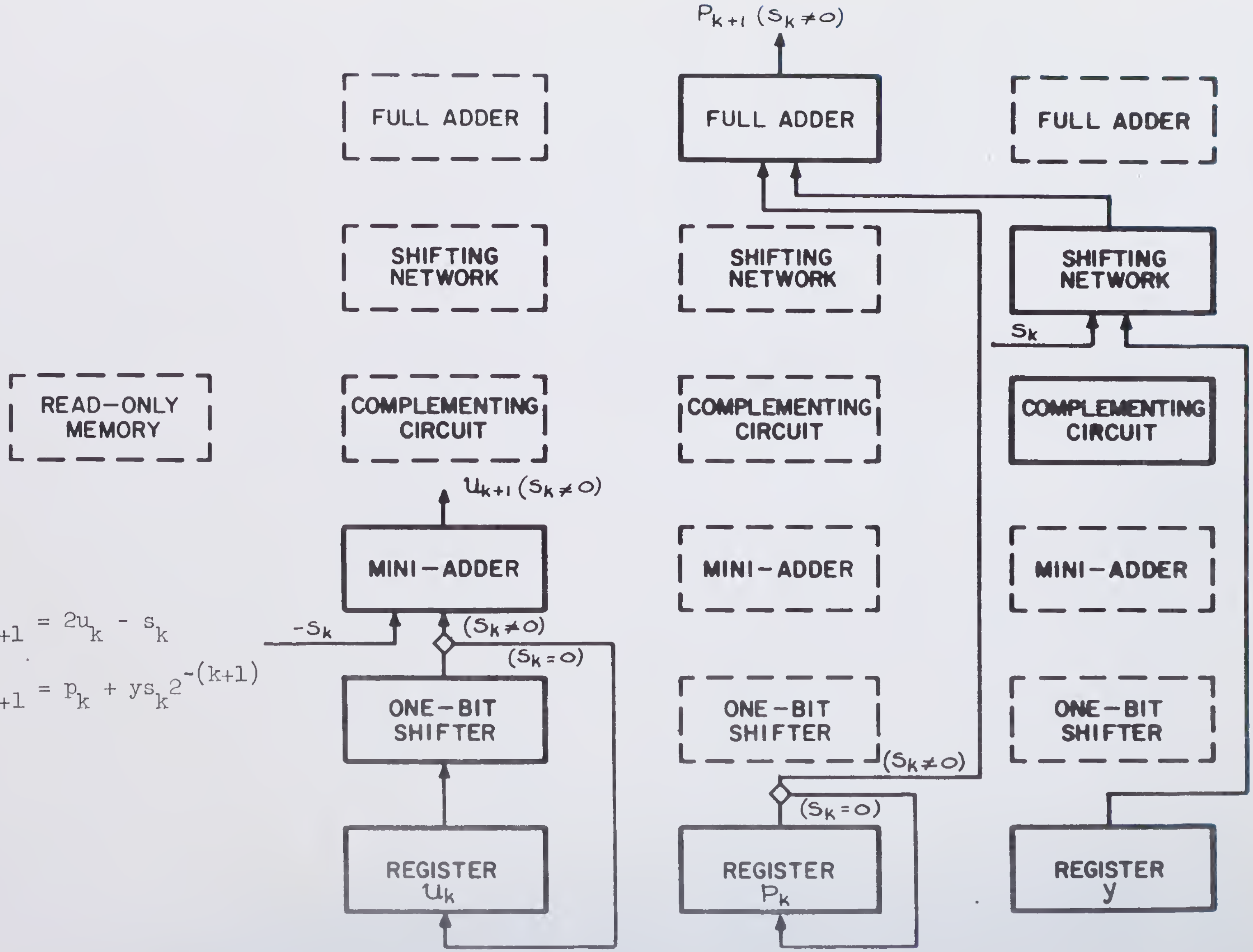
$$p_{k+1} = p_k + ys_k 2^{-k} \tag{4-2}$$

Except that a special register is required to hold the multiplicand y throughout the process, the hardware required to perform multiplication is virtually the same as that required for division.

A block diagram indicating the flow of information required in the implementation of (4-1) and (4-2) is shown in Figure 2.

4.6 Concluding remark

A multiplication algorithm using an implicit redundant recoding and yielding a shift average of approximately three has been proposed. It has been shown that this algorithm is compatible with the division algorithm in that both require virtually the same hardware configuration.



$$u_{k+1} = 2u_k - s_k$$

$$p_{k+1} = p_k + ys_k 2^{-(k+1)}$$

FIGURE 2. Block diagram for multiplication

5. THE ALGORITHM FOR NATURAL LOGARITHM

5.1 Basic algorithm

The algorithm to compute the natural logarithm of an operand $X > 0$ is very similar to the division algorithm discussed in Section 3; in fact, the normalization process is identical. Rather than forming the quotient in the second adder circuit, one merely forms the sum of a set of precomputed constants drawn from a register-speed read only memory [ROM], the ROM being the only piece of hardware, in addition to that required by division, required to implement this algorithm.

Given an operand $X = x \cdot 2^{\alpha}$, $x \in [1/2, 1)$, one may write,

$$\ell n\ X = \ell n\ x + \alpha\ \ell n\ 2.$$

The second term in this expression may be evaluated in one multiplication cycle time.

Let

$$\alpha = 2^N \sum_{i=0}^{N} m_i, \quad m_k = s_k 2^{-k}, \quad s_k = \{0, 1\}$$

where N determines the dynamic range of the machine, $(2^{2^{-N}}, 2^{2^{+N}})$. Typically $N \leq 10$; if $N = 10$ the dynamic range of the machine is $(2^{-1024}, 2^{+1024})$ or roughly $(10^{-300}, 10^{+300})$, sufficient for most problems. Since α contains relatively few bits (that is, normally N is considerably smaller than M), it is not necessary to recode α in order to speed up this multiplication; there is no advantage to completing this multiplication before the computation of $\ell n\ x$ is accomplished. A standard (non-redundant) multiplication process suffices.

$$\alpha\ \ell n\ 2 = 2^N [(\ell n\ 2) \sum_{i=0}^{N} m_i].$$

Letting

$$p_{k+1} = (\ell n\ 2) \sum_{i=0}^{k} m_i$$

the necessary recursion may be written,

$$p_{k+1} = p_k + s_k(\ell n\ 2)2^{-k}, \quad p_0 = 0$$

which is identical to recursion relation (4-2) with $y = \ell n\ 2$. Thus, one adder circuit configuration suffices to evaluate $\alpha\ \ell n\ 2$, providing only that the value of $\ell n\ 2$ is stored in the read only memory. Simultaneously and independently, the logarithm of the fractional part may be evaluated using two additional adder circuit configurations; three such configurations are required in all. A final addition is required, so that it takes one addition cycle time longer to compute the natural logarithm than to perform a division.

The computation of α ln 2 requires no further comment, so one may concentrate on an algorithm to compute $\ell n\ x$, $x \in [1/2, 1)$. As in the division algorithm of Section 3, one multiplies the operand x by a sequence of constants $\{\ell_i\}$, <u>conceptually</u> dividing x by the same continued product.

$$x = \frac{x \prod_{i=0}^{M} \ell_i}{\prod_{i=0}^{M} \ell_i}$$

where $\ell_k = d_k = 1 + 1/2\ s_k 2^{-k}$, $1 \leq k \leq M$, $\ell_0 = d_0$, the same constants as those chosen in the aforementioned division scheme. Then,

$$\ell n\ x = \ell n(x \prod_{i=0}^{M} \ell_i) - \ell n(\prod_{i=0}^{M} \ell_i)$$

$$= \ell n(x \prod_{i=0}^{M} \ell_i) + \sum_{i=0}^{M} (-\ell n\ \ell_i).$$

A power series expansion for $\ell n\ \delta$ is

$$\ell n\ \delta = (\delta - 1) - 1/2\,(\delta - 1)^2 + 1/3\,(\delta - 1)^3 - 1/4\,(\delta - 1)^4 + \ldots$$

$$[0 < \delta \leq 2].$$

It is known from Section 3.3 that

$$|x_{M+1} - 1| = |x \prod_{i=0}^{M} \ell_i - 1| \leq 3/8 \cdot 2^{-M}.$$

Then to machine accuracy (M bits),

$$\ell n\ (x \prod_{i=0}^{M} \ell_i) = 0$$

and

$$\ell n\ x = \sum_{i=0}^{M} (-\ell n\ \ell_i).$$

Providing that the values $\{-\ell n\ \ell_i\}$ are precomputed and stored in the read only memory, one may form $\ell n\ x$ by performing the normalization process of the division algorithm of Section 3 to choose the set $\{\ell_i\}$ with one adder configuration while forming the summation of the appropriate stored constants with a second adder configuration.

Note that when $s_k = 0$, $\ell_k = 1$ and $\ell n\ \ell_k = 0$, so that no addition need be performed in the either adder.

5.2 Choice of initialization step

Since the normalization process is identical to that of the proposed division, it suffices to choose $\ell_0 = d_0$. The value of $\ell n\ 2$ required by this choice was already to be stored in the memory.

An example is provided to illustrate the similarity of this algorithm to that for division.

<u>Example</u>: Given $x = 0.6$, $\ell n\ x = -0.51082562376599$. As indicated below, the algorithm produces an approximation to $\ell n\ x$ given by -0.51082562376644 which is in error by 0.45×10^{-12} matching the error bound derived in the first section.

In the table below, L_k represents the approximation to $\ell n\ x$ and x_k is the operand being normalized to unity.

TABLE 6

k	s_k	x_{k+1}	L_{k+1}
1	0	1.20000000000000	-0.69314718055995
2	-1	0.90000000000000	-0.40546510810816
3	1	1.01250000000000	-0.52324814376455
4	0	1.01250000000000	-0.52324814376455
5	0	1.01250000000000	-0.52324814376455
6	-1	0.99667968750000	-0.50749978679641
7	0	0.99667968750000	-0.50749978679641
8	1	1.00057296752930	-0.51139842721207
9	0	1.00057296752930	-0.51139842721207
10	0	1.00057296752930	-0.51139842721207
11	-1	1.00008440651000	-0.51091002671396
12	0	1.00008440651000	-0.51091002671396
13	0	1.00008440651000	-0.51091002671396
14	-1	1.00002336620198	-0.51084898969499
15	-1	0.99999284791078	-0.51081847165119

(Continued)

TABLE 6 (Continued)

k	s_k	x_{k+1}	L_{k+1}
16	0	0.99999284791078	-0.51081847165119
17	1	1.00000047725074	-0.51082610101662
18	0	1.00000047725074	-0.51082610101662
19	0	1.00000047725074	-0.51082610101662
20	0	1.00000047725074	-0.51082610101662
21	-1	1.00000000041336	-0.51082562417935
22	0	1.00000000041336	-0.51082562417935
23	0	1.00000000041336	-0.51082562417935
24	0	1.00000000041336	-0.51082562417935
25	0	1.00000000041336	-0.51082562417935
26	0	1.00000000041336	-0.51082562417935
27	0	1.00000000041336	-0.51082562417935
28	0	1.00000000041336	-0.51082562417935
29	0	1.00000000041336	-0.51082562417935
30	0	1.00000000041336	-0.51082562417935
31	-1	0.99999999994769	-0.51082562371368
32	0	0.99999999994769	-0.51082562371368
33	0	0.99999999994769	-0.51082562371368
34	1	1.00000000000590	-0.51082562377189
35	0	1.00000000000590	-0.51082562377189
36	0	1.00000000000590	-0.51082562377189
37	-1	0.99999999999863	-0.51082562376461
38	0	0.99999999999863	-0.51082562376461
39	1	1.00000000000044	-0.51082562376644
40	0	1.00000000000044	-0.51082562376644

5.3 Error bound

It has been shown that the algorithm provides M correct bits in the value of $\ell n\ x$, neglecting machine round-off.

5.4 Experimental estimate of speed

The value of $\ell n\ X$ can be computed in one addition cycle time beyond that required for division, or approximately $1 + M/3$ addition cycle times, on the average.

5.5 Implementation

The implementation of the evaluation of $\alpha\ \ell n\ 2$ is discussed in Section 5.1. The recursion relation for the normalization process is identical to (3-1) for division.

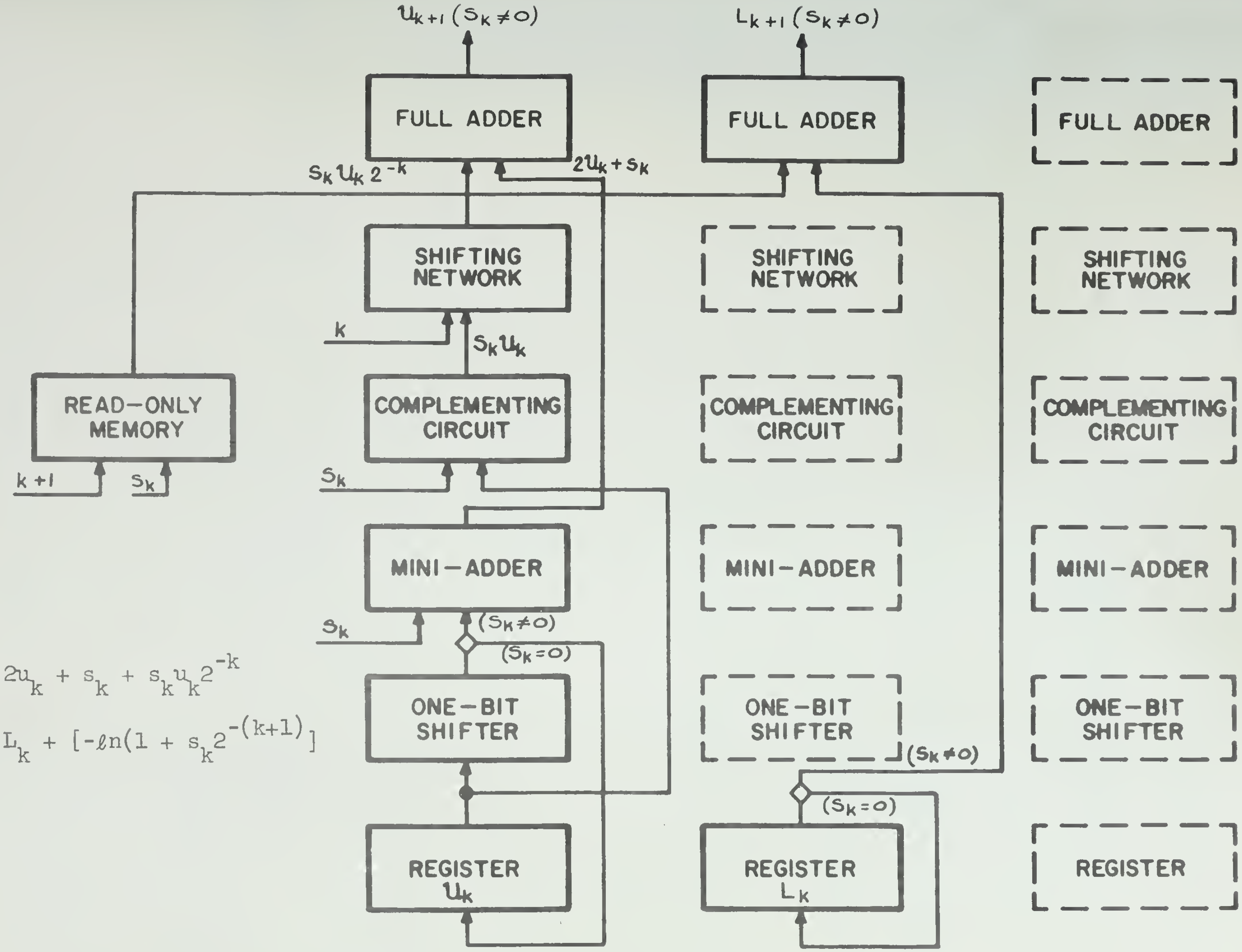
$$u_{k+1} = 2u_k + s_k + s_k u_k 2^{-k} \tag{5-1}$$

The second recursion relation is simply the continued summation of a set of stored constants.

$$\begin{aligned} L_{k+1} &= \sum_{i=0}^{k} (-\ell n\ \ell_i) \\ &= \sum_{i=0}^{k-1} (-\ell n\ \ell_i) + (-\ell n\ \ell_k) \\ &= L_k + [-\ell n\ (1 + s_k 2^{-(k+1)})] \end{aligned} \tag{5-2}$$

A block diagram indicating the flow of information is shown in Figure 3.

That only a portion of the set of precomputed constants $\{-\ell n\ (1 + s_k 2^{-(k+1)})\}$ need be explicitly stored in the ROM is easily seen. Consider again the power series expansion



$$u_{k+1} = 2u_k + s_k + s_k u_k 2^{-k}$$

$$L_{k+1} = L_k + [-\ell n(1 + s_k 2^{-(k+1)})]$$

FIGURE 3. Block diagram for logarithm

$$\ell n\ (1 + \delta) = \delta - 1/2\ \delta^2 + 1/3\ \delta^3 - 1/4\ \delta^4 + \ldots$$

$$[-1 < \delta \leq +1]$$

where $\delta = s_k 2^{-(k+1)}$. If $k > [\frac{M-3}{2}]$,*

$$\ell n\ (1 + \delta) = \delta = s_k 2^{-(k+1)}$$

to machine accuracy and the constants need not actually be stored.

5.6 Concluding remark

An algorithm for computing the natural logarithm has been proposed; the algorithm is virtually identical to the proposed division algorithm, except that a small ROM is required. The constants that must be precomputed and stored are

$$\ell n\ 2$$

$$-\ell n\ (1 \pm 2^{-(k+1)}) \qquad k = 1, 2, \ldots, [\frac{M-3}{2}].$$

A listing of approximate decimal equivalents of the required precomputed constants is given in Appendix B.

An algorithm for logarithm to any positive integer base could easily be formulated by the same procedure; new sets of precomputed constants would be required.

---

* The largest integer not greater than $(\frac{M-3}{2})$.

6. FIRST ALGORITHM FOR SQUARE ROOT

6.1 Basic algorithm

In Section 2.3 it was shown that four normalization square root algorithms are known; two of these are studied in this paper. The algorithm studied in this section is a multiplicative (continued product) formation of the quantity $y/\sqrt{x}$. The process is effectively the same as the proposed division algorithm, the only difference being that the multiplier constants $\{r_i\}$ are the squares of the multiplier constants chosen for division. Let

$$x = \frac{x \prod_{i=0}^{M} r_i}{\prod_{i=0}^{M} r_i}, \qquad x \in [1/2, 1)$$

where

$$r_k = (1 + 1/2\, s_k 2^{-k})^2, \quad 1 \leq k \leq M, \qquad s_k = \{\bar{1}, 0, 1\}$$

and $\{r_i\}$ is chosen such that

$$x \prod_{i=0}^{M} r_i \cong 1.$$

Then,

$$\prod_{i=0}^{M} r_i \cong \frac{1}{x}$$

and

$$y \prod_{i=0}^{M} (1 + 1/2\, s_i 2^{-i}) \cong \frac{y}{\sqrt{x}} .$$

The recursion relation for the normalization of the operand x is complicated by the fact that the multipliers have three terms rather than two.

$$x_{k+1} = x_0 \prod_{i=0}^{k} r_i \qquad x_0 = x$$

$$= (x_0 \prod_{i=0}^{k-1} r_i)(1 + 1/2\ s_k 2^{-k})^2$$

$$= x_k(1 + s_k 2^{-k} + s_k^2\ 2^{-2(k+1)}) \tag{6-1}$$

$$R_{k+1} = R_0 \prod_{i=0}^{k} \sqrt{r_i} \qquad R_0 = y$$

$$= (R_0 \prod_{i=0}^{k-1} \sqrt{r_i}\ )(1 + 1/2\ s_k 2^{-k})$$

$$= R_k\ (1 + s_k 2^{-(k+1)}). \tag{6-2}$$

The multiplier constants for the division algorithm are

$$d_k = 1 + 1/2\ s_k 2^{-k}$$

whereas those chosen in this square root algorithm are

$$r_k = 1 + s_k 2^{-k} + s_k^2\ 2^{-2(k+1)}.$$

The dominant terms (other than unity) in the multipliers differ by a factor of two; for this reason, the comparison constants must also differ by a factor of two to achieve the same recoding. A simple change in notation or a comparable change in the implementation transformation can overcome this minor discrepancy; the latter is chosen as being conceptually neater; in practice, the two are the same. Thus, in terms of the partially normalized operand x_k, the selection rules are chosen as follows.

$$r_k = (1 + 1/2\ s_k 2^{-k})^2$$

$$s_k = \begin{cases} 1 & \text{if} \quad x_k < 1 - 3/4 \cdot 2^{-k} \\ 0 & \quad\text{otherwise} \\ \bar{1} & \text{if} \quad x_k \geq 1 + 3/4 \cdot 2^{-k}. \end{cases}$$

The implementation transformation in the division algorithm is given by

$$u_k = 2^k(x_k - 1)$$

whereas the implementation transformation in this algorithm is taken as

$$u_k = 2^{k-1}(x_k - 1)$$

so that $|u_k| < 1$, as is shown in Section 6.3. The selection rules, in terms of u_k, may thus be written in exactly the same form as that for division:

$$s_k = \begin{cases} 1 & \text{if} \quad u_k < -3/8 \\ 0 & \quad\text{otherwise} \\ \bar{1} & \text{if} \quad u_k \geq +3/8. \end{cases}$$

6.2 Choice of initialization step

The initialization step comparable to that chosen for the division algorithm is $r_0 = d_0^2$, that is,

$$r_0 = \begin{cases} 4 & \text{if} \quad 1/4 \leq x_0 < 3/4 \\ 1 & \text{if} \quad 3/4 \leq x_0 < 1 \end{cases}$$

which leaves $x_1 = x_0 r_0$ in the range $[3/4, 3)$ and $u_1 = x_1 - 1 \in [-1/4, 2)$, outside the desired range, $(-1, +1)$. Merely changing the initial comparison constant provides a more acceptable range.

$$r_0 = \begin{cases} 4 & \text{if} \quad 1/4 \leq x_0 < 1/2 \\ 1 & \text{if} \quad 1/2 \leq x_0 < 1. \end{cases}$$

Thus, $x_1 \in [1/2, 2)$, $u_1 \in [-1/2, 1)$. It is shown in the next section that this choice of initialization leads to a convergent algorithm.

Example: An example computing $0.5/\sqrt{0.6} = 0.64549722436790$ is tabulated below. It may be seen that the algorithm produces an approximation which is in error by 0.11×10^{-12}, well within the error bound of 0.68×10^{-12} derived in the next section.

TABLE 7

k	s_k	x_{k+1}	R_{k+1}
1	1	0.93750000000000	0.62500000000000
2	0	0.93750000000000	0.62500000000000
3	0	0.93750000000000	0.62500000000000
4	1	0.99700927734375	0.64453125000000
5	0	0.99700927734375	0.64453125000000
6	0	0.99700927734375	0.64453125000000
7	0	0.99700927734375	0.64453125000000
8	1	1.00090764812194	0.64579010009766
9	0	1.00090764812194	0.64579010009766
10	-1	0.99993043788180	0.64547477290034
11	0	0.99993043788180	0.64547477290034
12	0	0.99993043788180	0.64547477290034
13	0	0.99993043788180	0.64547477290034
14	1	0.99999146972357	0.64549447122715
15	0	0.99999146972357	0.64549447122715

(Continued)

TABLE 7 (Continued)

k	s_k	x_{k+1}	R_{k+1}
16	0	0.99999146972357	0.64549447122715
17	1	0.99999909906757	0.64549693359315
18	0	0.99999909906757	0.64549693359315
19	0	0.99999909906757	0.64549693359315
20	1	1.00000005274126	0.64549724139007
21	0	1.00000005274126	0.64549724139007
22	0	1.00000005274126	0.64549724139007
23	0	1.00000005274126	0.64549724139007
24	-1	0.99999999313661	0.64549722215275
25	0	0.99999999313661	0.64549722215275
26	0	0.99999999313661	0.64549722215275
27	1	1.00000000058719	0.64549722455742
28	0	1.00000000058719	0.64549722455742
29	0	1.00000000058719	0.64549722455742
30	0	1.00000000058719	0.64549722455742
31	-1	1.00000000012153	0.64549722440713
32	0	1.00000000012153	0.64549722440713
33	-1	1.00000000000511	0.64549722436955
34	0	1.00000000000511	0.64549722436955
35	0	1.00000000000511	0.64549722436955
36	0	1.00000000000511	0.64549722436955
37	0	1.00000000000511	0.64549722436955
38	-1	1.00000000000148	0.64549722436838
39	-1	0.99999999999657	0.64549722436779
40	0	0.99999999999657	0.64549722436779

6.3 Error bound

It has already been shown that

$$x_0 \in [1/4,\ 1), \quad u_0 \in [-3/8,\ 0)$$

$$x_1 \in [1/2,\ 2), \quad u_1 \in [-1/2,\ 1).$$

Next let us find the range of x_2. The selection rule for the first $(k = 1)$ step is given by,

$$r_1 = 1 + 1/2\ s_1 + 1/16\ s_1^2$$

$$s_1 = \begin{cases} 1 & \text{if} \quad 1/2 \leq x_1 < 5/8 \\ 0 & \text{if} \quad 5/8 \leq x_1 < 11/8 \\ \bar{1} & \text{if} \quad 11/8 \leq x_1 < 2. \end{cases}$$

The first range,

$$[1/2,\ 5/8)$$

maps onto

$$(1 + 1/2 + 1/16) \cdot [1/2,\ 5/8)$$

or

$$[25/32,\ 125/128).$$

The second range,

$$[5/8,\ 11/8)$$

maps onto itself.

The third range,

$$[11/8,\ 2)$$

maps onto

$$(1 - 1/2 + 1/16) \cdot [11/8,\ 2)$$

or

$$[99/128,\ 9/8).$$

Hence,

$$x_2 \in [5/8,\ 11/8), \quad u_2 \in [-3/4,\ +3/4)$$

so that x_2 lies in the middle range of the selection rules for s_1 and $|u_2| < 1$. It is easy to show that

$$|x_{M+1} - 1| \leq 3/4 \cdot 2^{-M}, \quad |u_k| < 1 \text{ for all } k.$$

<u>Hypothesis</u>: For $k \geq 2$, $x_k \in [1 - 3/4 \cdot 2^{-(k-1)},\ 1 + 3/4 \cdot 2^{-(k-1)})$.
The hypothesis has been shown to be true for $k = 2$. The induction proof for the behavior for $k > 2$ is virtually identical to that in Section 3.3 for division, except for the perturbation caused by the second order term in the multiplier.

<u>Proof</u>: For some $k \geq 2$, $x_k \in [1 - 3/2 \cdot 2^{-k},\ 1 + 3/2 \cdot 2^{-k})$. For all $k \geq 2$,

$$r_k = 1 + s_k 2^{-k} + 1/4\ s_k^2\ 2^{-2k}$$

$$s_k = \begin{cases} 1 & \text{if} \quad x_k < 1 - 3/4 \cdot 2^{-k} \\ 0 & \qquad \text{otherwise} \\ \bar{1} & \text{if} \quad x_k \geq 1 + 3/4 \cdot 2^{-k}. \end{cases}$$

The first range,

$$[1 - 3/2 \cdot 2^{-k},\ 1 - 3/4 \cdot 2^{-k})$$

maps onto

$$[1 - 2^{-k}(1/2 + 5/4 \cdot 2^{-k} + 3/8 \cdot 2^{-2k}),$$
$$1 + 2^{-k}(1/4 - 1/2 \cdot 2^{-k} - 3/16 \cdot 2^{-2k})).$$

But for $k > 2$,

$$1/2 + 5/4 \cdot 2^{-k} + 3/8 \cdot 2^{-2k} \leq 339/512 < 3/4$$

so here

$$x_{k+1} \in [1 - 3/4 \cdot 2^{-k},\ \ 1 + 3/4 \cdot 2^{-k}).$$

The middle range,

$$[1 - 3/4 \cdot 2^{-k},\ \ 1 + 3/4 \cdot 2^{-k})$$

maps onto itself.

The last range,

$$[1 + 3/4 \cdot 2^{-k},\ \ 1 + 3/2 \cdot 2^{-k})$$

maps onto

$$[1 - 2^{-k}(1/4 + 1/2 \cdot 2^{-k} - 3/16 \cdot 2^{-2k}),$$
$$1 + 2^{-k}(1/2 - 5/4 \cdot 2^{-k} + 3/8 \cdot 2^{-2k}))$$

which clearly lies within the desired range. Thus,

$$|x_{M+1} - 1| \leq 3/4 \cdot 2^{-M}$$

and

$$|u_k| < 1 \qquad \text{for all } k.$$

Let

$$\lambda = \prod_{i=0}^{M} \sqrt{r_i}$$

so that

$$x\,\lambda^2 = x_{M+1}$$

and

$$|x\,\lambda^2 - 1| \leq 3/4 \cdot 2^{-M}.$$

Then

$$x(\lambda + \frac{1}{\sqrt{x}})\,|\lambda - \frac{1}{\sqrt{x}}\,| \leq 3/4 \cdot 2^{-M}.$$

Let ϵ be defined by

$$\lambda = \frac{1+\epsilon}{\sqrt{x}}$$

where ϵ represents a relative error in the approximation to $1/\sqrt{x}$; $|\epsilon| \ll 1$.

Then,

$$x\left(\frac{2+\epsilon}{\sqrt{x}}\right)\left|\frac{\epsilon}{\sqrt{x}}\right| \leq 3/4 \cdot 2^{-M}$$

or

$$(2+\epsilon)|\epsilon| \leq 3/4 \cdot 2^{-M}.$$

Since $|\epsilon| \ll 1$, one may write approximately

$$|\epsilon| \leq 3/8 \cdot 2^{-M}.$$

Thus,

$$\left|\lambda - \frac{1}{\sqrt{x}}\right| \leq \frac{1}{\sqrt{x}}(3/8 \cdot 2^{-M})$$

or

$$\left|\lambda - \frac{1}{\sqrt{x}}\right| \leq 3/4 \cdot 2^{-M}.$$

Finally, since $y < 1$,

$$y\left|\lambda - \frac{1}{\sqrt{x}}\right| < 3/4 \cdot 2^{-M},$$

that is, in order to guarantee M correct bits in the value of $y/\sqrt{x}$, one must perform M + 1 steps beyond the initialization.

6.4 Experimental estimate of speed

The Monte Carlo estimate of the mean probability of a zero is 0.669, with a corresponding shift average of 3.04.

6.5 Implementation

Rewriting recursions (6-1) and (6-2) under the transformation $u_k = 2^{k-1}(x_k - 1)$, one obtains,

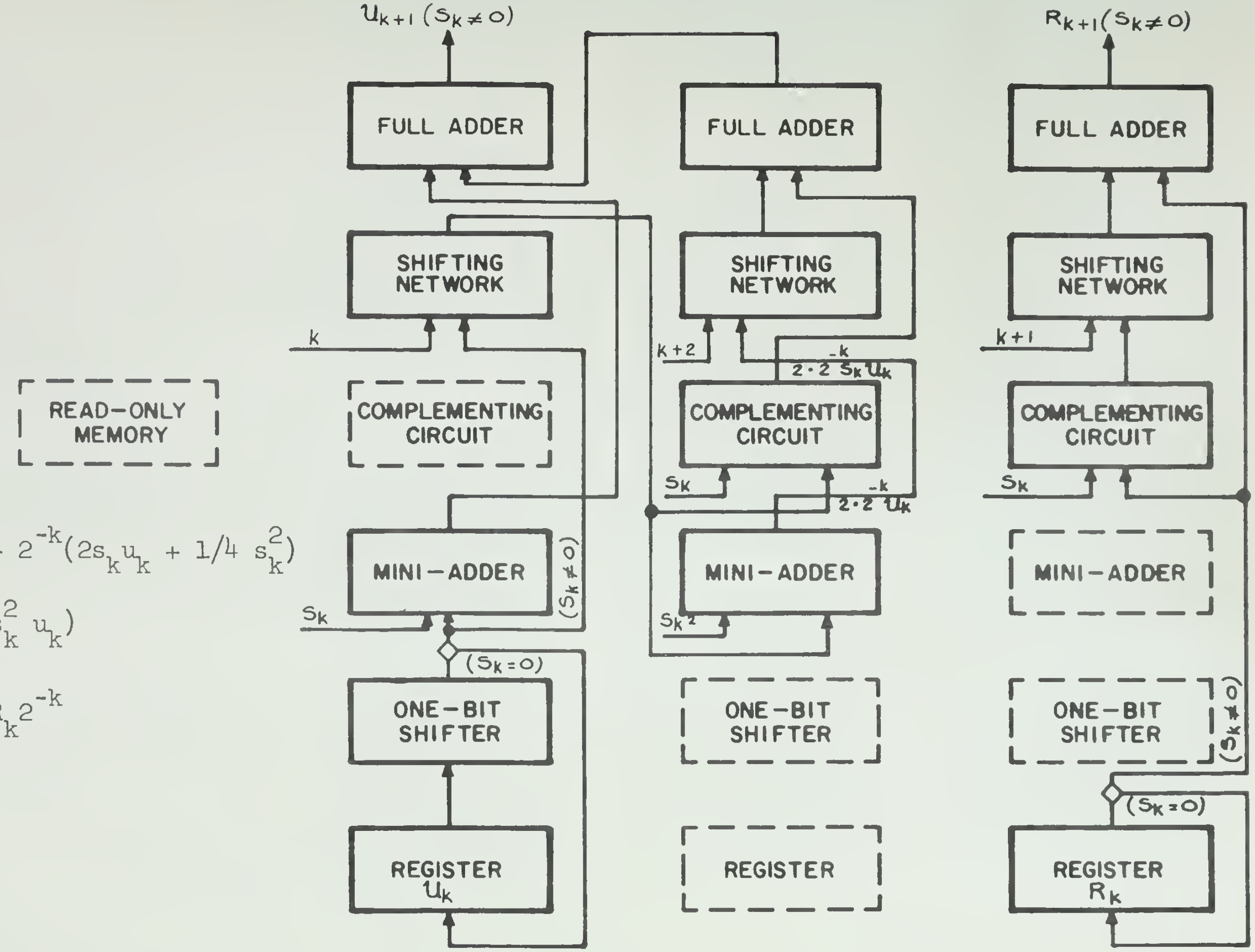
$$u_{k+1} = (2u_k + s_k) + 2^{-k}(2s_k u_k + 1/4\ s_k^2) + 2^{-2k}(1/2\ s_k^2 u_k),$$
$$u_0 = 1/2(x-1) \qquad (6\text{-}3)$$

$$R_{k+1} = R_k + 1/2\ s_k R_k 2^{-k}, \qquad R_0 = y. \qquad (6\text{-}4)$$

Figures 4, 5, and 6 show possible hardware configurations to implement these recursion relations.

6.6 Concluding remark

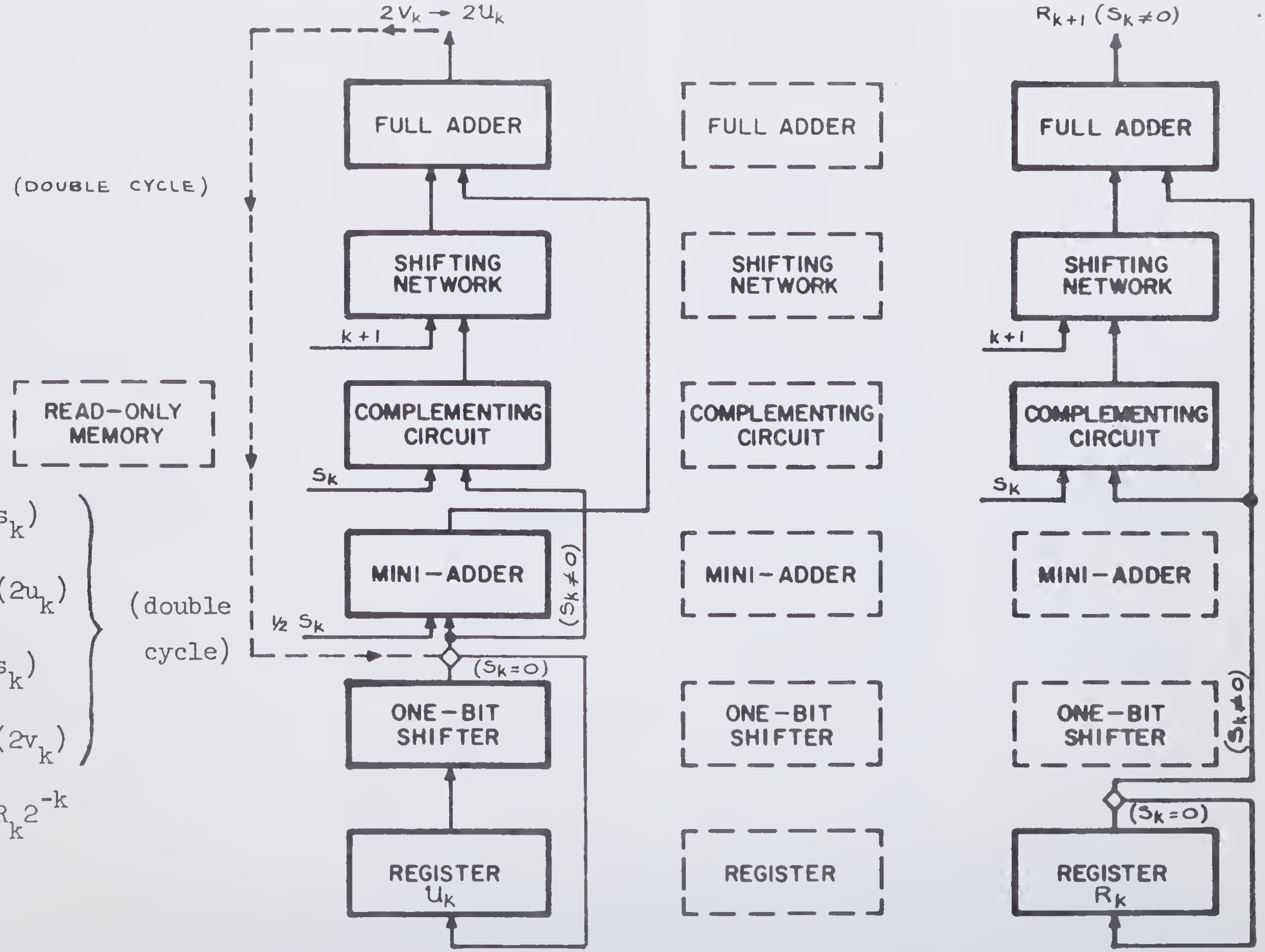
Further comments about this algorithm are included in Section 7.6 where a comparison of the two square root algorithms is made.



$$u_{k+1} = (2u_k + s_k) + 2^{-k}(2s_k u_k + 1/4\ s_k^2)$$
$$+\ 2^{-2k}(1/2\ s_k^2\ u_k)$$

$$R_{k+1} = R_k + 1/2\ s_k R_k 2^{-k}$$

FIGURE 4. First block diagram for square root



$$\left.\begin{aligned} 2v_k &= (2u_k + 1/2\ s_k) \\ &\quad + 2^{-(k+1)} s_k (2u_k) \\ u_{k+1} &= (2v_k + 1/2\ s_k) \\ &\quad + 2^{-(k+1)} s_k (2v_k) \end{aligned}\right\} \text{(double cycle)}$$

$$R_{k+1} = R_k + 1/2\ s_k R_k 2^{-k}$$

FIGURE 5. Second block diagram for square root

$$\left.\begin{aligned} 2v_k &= (2u_k + 1/2\ s_k) \\ &\quad + 2^{-(k+1)} s_k (2u_k) \\ u_{k+1} &= (2v_k + 1/2\ s_k) \\ &\quad + 2^{-(k+1)} s_k (2v_k) \end{aligned}\right\} \text{(one pass)}$$

$$R_{k+1} = R_k + 1/2\ s_k R_k 2^{-k}$$

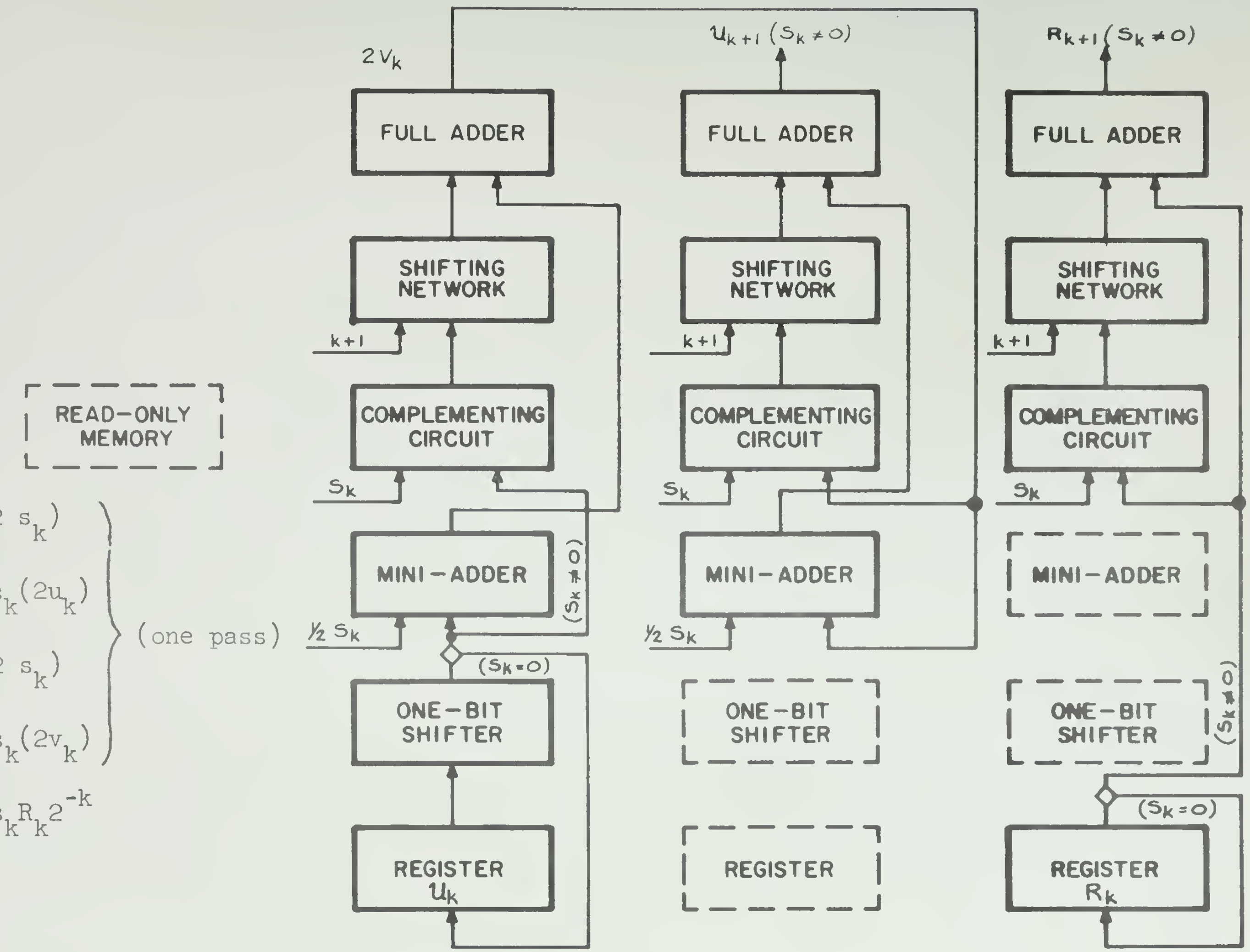


FIGURE 6. Third block diagram for square root

7. SECOND ALGORITHM FOR SQUARE ROOT

7.1 Basic algorithm

Let us consider in this section an additive square root algorithm which can be performed in approximately the same amount of time as the division algorithm and with essentially the same hardware, but which requires a scaling of the comparison constants to achieve a minimal recoding. It is somewhat less general than the algorithm studied in the last section in that the root itself, rather than $y/\sqrt{x}$, is evaluated.

The normalization process one wishes to carry out is

$$\gamma_{k+1} = \gamma_k - r_k$$

where

$$\gamma_0 = \sqrt{x}$$

$$r_k = 1/2\ s_k 2^{-k}$$

$$s_k = \begin{cases} \bar{1} & \text{if} \quad \gamma_k < -c \cdot 2^{-k} \\ 0 & \text{otherwise} \\ 1 & \text{if} \quad \gamma_k \geq +c \cdot 2^{-k} \end{cases}$$

$$c \in [1/3,\ 5/12)$$

so that

$$\gamma_{M+1} \cong 0$$

$$R_{M+1} = \sum_{i=0}^{M} r_i \cong \sqrt{x}.$$

One would achieve an asymptotic shift average of three with this recoding. While this is a conceptually neat algorithm, it cannot be carried out in practice in this form since $\sqrt{x}$ is unknown and the recursion cannot be

initialized. Rather, one must perform recursions with respect to a known quantity such as

$$x_{k+1} = x_0 - \left[\sum_{i=0}^{k} r_i \right]^2, \qquad x_0 = x$$

where

$$r_k = 1/2\ s_k 2^{-k}$$

$$s_k = \begin{cases} \bar{1} & \text{if} \quad x_k < -c' \cdot 2^{-k} \\ 0 & \text{otherwise} \\ 1 & \text{if} \quad x_k \geq +c' \cdot 2^{-k}. \end{cases}$$

One must thus find a suitable relationship between c' and c that yields a minimal recoding. Now,

$$\gamma_{k+1} = \sqrt{x} - R_{k+1}$$

$$x_{k+1} = x - R_{k+1}^2$$

so

$$x_{k+1} = \gamma_{k+1}(\sqrt{x} + R_{k+1})$$

$$\cong \gamma_{k+1}(2\sqrt{x}).$$

The problem thus becomes one of choosing a convenient value for $c' = 2\sqrt{x}\,c$, $c \in [1/3, 5/12)$. As an example, suppose $\sqrt{x} = 1/2$ $(x = 1/4)$; then $c' = c = 3/8$ is the most convenient choice; if $\sqrt{x} = 1$ $(x = 1)$, then $c' = 2c = 3/4$ is convenient. Metze[1] solved a similar scaling problem for the SRT division; he showed that at least four regions are needed to cover the range, whose endpoints are in the ratio 2/1, since $(5/4)^3 < 2$, $(5/4)^4 > 2$.

To reduce the precision of the comparisons, five regions are recommended. Convenient choices for comparison constants for the various ranges of x are listed in Table 8.

TABLE 8

x	$\sqrt{x}$	Comparison Constant c'
$[\frac{1}{4}, \frac{5}{16})$	$[0.500, 0.558)$	$\frac{3}{8}$
$[\frac{5}{16}, \frac{3}{8})$	$[0.558, 0.612)$	$\frac{7}{16}$
$[\frac{3}{8}, \frac{9}{16})$	$[0.612, 0.750)$	$\frac{1}{2}$
$[\frac{9}{16}, \frac{7}{8})$	$[0.750, 0.935)$	$\frac{5}{8}$
$[\frac{7}{8}, 1)$	$[0.935, 1)$	$\frac{3}{4}$

7.2 Choice of initialization step

Let us consider, for a moment, the recursions necessary to implement this algorithm.

$$x_{k+1} = x_0 - \left[\sum_{i=0}^{k} r_i\right]^2 \qquad\qquad x_0 = x$$

$$= x_0 - \left[\sum_{i=0}^{k-1} r_i + r_k\right]^2$$

$$= x_0 - \left[\sum_{i=0}^{k-1} r_i\right]^2 - s_k 2^{-k} R_k - s_k^2 2^{-2(k+1)}$$

$$= x_k - s_k R_k 2^{-k} - s_k^2 2^{-2(k+1)} \tag{7-1}$$

$$R_{k+1} = \sum_{i=0}^{k} r_i \qquad\qquad R_0 = 0$$

$$= R_k + s_k 2^{-(k+1)} \tag{7-2}$$

Rewriting (7-1) under the transformation $u_k = 2^{(k-2)} x_k$, chosen to force $|u_k| < 1$,

$$u_{k+1} = 2u_k - 1/2(s_k R_k + s_k^2\, 2^{-(k+2)}), \qquad u_0 = 1/4\, x_0. \tag{7-3}$$

If it can be guaranteed that R_k has a zero in bit position $(k + 2)$ then the single bit represented by $s_k^2\, 2^{-(k+2)}$ can simply be inserted in $s_k R_k$ and the value $(s_k R_k + s_k^2\, 2^{-(k+2)})$ can be shifted, complemented, and added to $2u_k$. If the initialization constant r_0 is a low precision number, and if radix complement notation is used for negative numbers, $s_k R_k$ will indeed have a zero in bit position $(k + 2)$ as desired. This minimizes the delay time caused by the mini-adder.

Since the comparison constant for the algorithm is a function of the operand, the initialization constant r_0 is also allowed to be a function of the operand.

TABLE 9

x	r_0
$[\frac{1}{4}, \frac{5}{16})$	$\frac{1}{2}$
$[\frac{5}{16}, \frac{3}{8})$	$\frac{9}{16}$
$[\frac{3}{8}, \frac{9}{16})$	$\frac{5}{8}$
$[\frac{9}{16}, \frac{7}{8})$	$\frac{3}{4}$
$[\frac{7}{8}, 1)$	1

It is shown in the next section that these choices for initialization lead to a convergent algorithm.

An example is listed below.

Example: If $x = 0.6$, then $\sqrt{x} = 0.77459666924148$. In 40 steps, the algorithm produces an approximation to $\sqrt{x}$ which is in error by 0.6×10^{-13}, well within the worst case error bound of 0.45×10^{-12} (derived in the next section).

TABLE 10

k	s_k	x_{k+1}	R_{k+1}
1	0	0.03750000000000	0.75000000000000
2	0	0.03750000000000	0.75000000000000
3	0	0.03750000000000	0.75000000000000
4	0	0.03750000000000	0.75000000000000
5	1	0.01381835937500	0.76562500000000
6	1	0.00179443359375	0.77343750000000
7	0	0.00179443359375	0.77343750000000
8	0	0.00179443359375	0.77343750000000
9	1	0.00028285980225	0.77441406250000
10	0	0.00028285980225	0.77441406250000
11	0	0.00028285980225	0.77441406250000
12	1	0.00009377896786	0.77453613281250
13	1	-0.00000077262521	0.77459716796875
14	0	-0.00000077262521	0.77459716796875
15	0	-0.00000077262521	0.77459716796875
16	0	-0.00000077262521	0.77459716796875
17	0	-0.00000077262521	0.77459716796875
18	0	-0.00000077262521	0.77459716796875
19	0	-0.00000077262521	0.77459716796875
20	-1	-0.00000003391201	0.77459669113159

(Continued)

TABLE 10 (Continued)

k	s_k	x_{k+1}	R_{k+1}
21	0	-0.0000003391201	0.77459669113159
22	0	-0.0000003391201	0.77459669113159
23	0	-0.0000003391201	0.77459669113159
24	0	-0.0000003391201	0.77459669113159
25	-1	-0.0000001082723	0.77459667623043
26	-1	0.0000000071516	0.77459666877985
27	0	0.0000000071516	0.77459666877985
28	0	0.0000000071516	0.77459666877985
29	0	0.0000000071516	0.77459666877985
30	1	-0.0000000000624	0.77459666924551
31	0	-0.0000000000624	0.77459666924551
32	0	-0.0000000000624	0.77459666924551
33	0	-0.0000000000624	0.77459666924551
34	0	-0.0000000000624	0.77459666924551
35	0	-0.0000000000624	0.77459666924551
36	0	-0.0000000000624	0.77459666924551
37	-1	-0.0000000000060	0.77459666924187
38	0	-0.0000000000060	0.77459666924187
39	0	-0.0000000000060	0.77459666924187
40	-1	0.0000000000010	0.77459666924142

7.3 Error bound

In this section it is shown that the error in the approximation to the root is bounded by $2^{-(M+1)}$, so that M correct bits in the root are produced in M steps beyond the initialization.

The first step of the proof consists of producing, by direct computation, bounds on the first few approximations to $\sqrt{x}$. The proof is then

completed by induction. Recall,

$$\gamma_k = \sqrt{x} - R_k, \qquad \gamma_0 = \sqrt{x} \in [1/2,\ 1).$$

Table 9 is then completed to yield Table 11.

TABLE 11

x	$\sqrt{x}$	r_o	$\gamma_1 = \sqrt{x} - r_o$	$x_1 = x - r_o^2$
$[\frac{1}{4}, \frac{5}{16})$	[0.500, 0.558)	$\frac{1}{2}$	[0, +0.058)	$[0, \frac{1}{16})$
$[\frac{5}{16}, \frac{3}{8})$	[0.558, 0.612)	$\frac{9}{16}$	[-0.0045, +0.0495)	$[-\frac{1}{256}, \frac{15}{256})$
$[\frac{3}{8}, \frac{9}{16})$	[0.612, 0.750)	$\frac{5}{8}$	[-0.013, +0.125)	$[-\frac{1}{64}, \frac{11}{64})$
$[\frac{9}{16}, \frac{7}{8})$	[0.750, 0.935)	$\frac{3}{4}$	[0, +0.185)	$[0, \frac{5}{16})$
$[\frac{7}{8}, 1)$	[0.935, 1)	1	[-0.065, 0)	$[-\frac{1}{8}, 0)$

Hence,

$$\gamma_1 \in [-0.065,\quad +0.185).$$

By looking at the selection rules for the first ($k = 1$) step, one can see that $s_1 = 0$, $r_1 = 0$, independent of x (a virtue of the initialization chosen). Then

$$\gamma_2 \in [-0.065,\quad +0.185).$$

<u>Hypothesis</u>: $|\gamma_k| \leq 2^{-k}$ for all k. The hypothesis has been shown to be true for $k = 0, 1, 2$. Assume $|\gamma_k| \leq 2^{-k}$ for some k and show $|\gamma_{k+1}| \leq 2^{-(k+1)}$.

Proof: For the k^{th} step,

$$r_k = 1/2\ s_k 2^{-k}$$

$$s_k = \begin{cases} \bar{1} & \text{if} \quad x_k < -c' \cdot 2^{-k} \\ 0 & \quad\quad \text{otherwise} \\ 1 & \text{if} \quad x_k \geq +c' \cdot 2^{-k} \end{cases}$$

where $c' = 2\sqrt{x}\ c$, $\quad c \in [\ 1/3,\ 5/12)$.

Range 1: Suppose $x_k < -c' \cdot 2^{-k}$ so that $s_k = \bar{1}$, $r_k = -\ 1/2 \cdot 2^{-k}$. Then,

$$\gamma_{k+1} = \gamma_k - r_k$$

$$= \gamma_k + 1/2 \cdot 2^{-k}.$$

Clearly the signs of x_k and γ_k must agree, so

$$-2^{-k} \leq \gamma_k < 0$$

and thus,

$$|\gamma_{k+1}| \leq 2^{-(k+1)}.$$

Range 2: Suppose $-c' \cdot 2^{-k} \leq x_k < c' \cdot 2^{-k}$ so that $s_k = 0$, $r_k = 0$. Now,

$$\gamma_{k+1} = \frac{x_k}{2\sqrt{x} - \gamma_k}$$

$$= \frac{x_k}{2\sqrt{x}\,(1 - \frac{\gamma_k}{2\sqrt{x}})}$$

$$|\gamma_{k+1}| \leq \frac{c' \cdot 2^{-k}}{2\sqrt{x}\,(1 - \frac{\gamma_k}{2\sqrt{x}})}$$

$$|\gamma_{k+1}| \leq \frac{2\sqrt{x} \cdot c \cdot 2^{-k}}{2\sqrt{x}\,(1 - \frac{\gamma_k}{2\sqrt{x}})}$$

$$|\gamma_{k+1}| \leq \frac{5/12 \cdot 2^{-k}}{(1 - 2^{-k})} .$$

For $k \geq 3$,

$$\frac{1}{1 - 2^{-k}} \leq 8/7$$

so

$$|\gamma_{k+1}| \leq 5/12 \cdot 8/7 \cdot 2^{-k} < 2^{-(k+1)}.$$

Range 3: Suppose $x_k \geq c' \cdot 2^{-k}$ so that $s_k = 1$, $r_k = 1/2 \cdot 2^{-k}$. Then,

$$\gamma_{k+1} = \gamma_k - r_k$$

$$= \gamma_k - 1/2 \cdot 2^{-k}.$$

Since the signs of x_k and γ_k agree,

$$0 < \gamma_k \leq +2^{-k}$$

and thus

$$|\gamma_{k+1}| \leq 2^{-(k+1)}.$$

Hence, for all k,

$$|\gamma_k| = |R_k - \sqrt{x}| \leq 2^{-k}.$$

Q.E.D.

Next it is shown that $|u_k| < 1$ for all k. Recall $u_k = 2^{(k-2)}x_k$. Since $x_0 \in [1/4, 1)$, $u_0 \in [1/16, 1/4)$. Also,

$$
\begin{aligned}
|x_k| &= |R_k^2 - x| \\
&= |R_k - \sqrt{x}|(R_k + \sqrt{x}) \\
&\leq 2^{-k}(-\sqrt{x} + 2^{-k} + \sqrt{x}) \\
&\leq 2^{-k}(2\sqrt{x} + 2^{-k}) \\
|u_k| &\leq 1/4\ (2\sqrt{x} + 2^{-k}) \\
&\leq 1/4\ (2 + 1/2) \quad \text{for } k \geq 1.
\end{aligned}
$$

Hence, $|u_k| < 1$ for all k.

7.4 Experimental estimate of speed

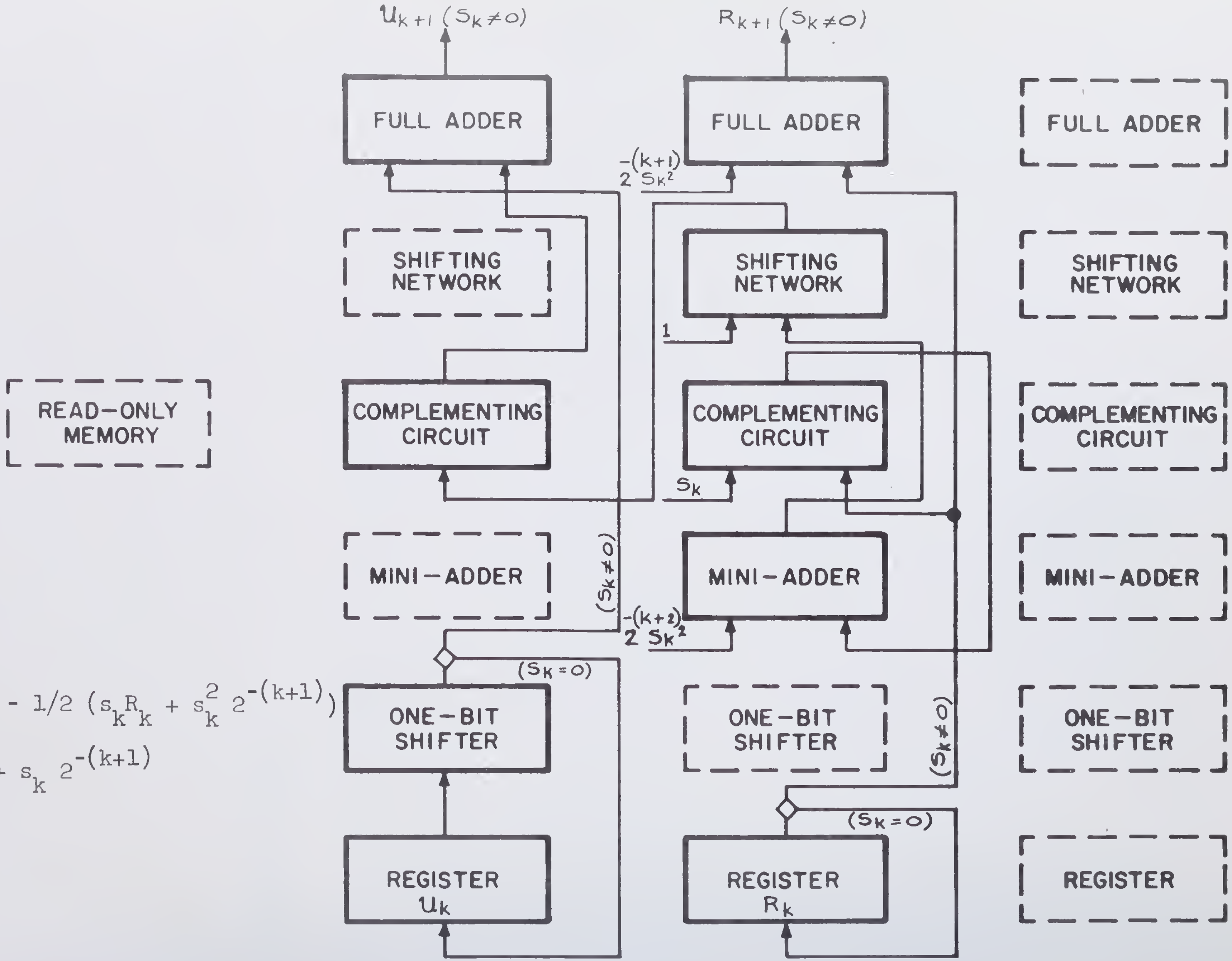
The Monte Carlo estimate of the mean probability of a zero is 0.661, with a corresponding shift average of 2.96.

7.5 Implementation

The recursion formulas for implementation are (7-2) and (7-3). Figure 7 shows the corresponding block diagram.

7.6 Concluding remark

Two square root algorithms have been studied in detail, a multiplicative scheme in Section 6 and an additive scheme in this section. The multiplicative scheme clearly requires more hardware to achieve a speed comparable to that of the additive scheme; it cannot achieve equality of speed. It thus appears that the multiplicative algorithm should be discarded in favor of the additive algorithm, and this is probably true in a strictly binary implementation. However, the algorithm of Section 6 offers two redeeming features: first, the comparison constants need not be scaled; second, the algorithm easily lends itself to a higher radix implementation, which the algorithm of Section 7 does not.



$$u_{k+1} = 2u_k - 1/2\,(s_k R_k + s_k^2\, 2^{-(k+1)})$$

$$R_{k+1} = R_k + s_k\, 2^{-(k+1)}$$

FIGURE 7. Fourth block diagram for square root

REFERENCE

1 G. A. Metze, "A Class of Binary Divisions Yielding Minimally Represented Quotients," IRE Transactions on Electronic Computers, EC-11:6: 761-764, December, 1962.

8. THE ALGORITHM FOR EXPONENTIAL

8.1 Basic algorithm

The exponential e^X may be evaluated in three multiplication cycle times, two of which are required to identify a convenient operand which may be normalized to zero. Let

$$
\begin{aligned}
e^X &= e^{X \log_2 e \, \log_e 2} \\
&= e^{(I+f) \ln 2}
\end{aligned}
$$

where

$$
\begin{aligned}
X \log_2 e &= I + f \\
I &= \text{integer} \\
-1 &< f < +1.
\end{aligned}
$$

Then,

$$
\begin{aligned}
e^X &= e^{I \,\ell n\, 2} \, e^{f \,\ell n\, 2} \\
&= 2^I \, e^x
\end{aligned}
$$

where

$$
\begin{aligned}
x &= f \,\ell n\, 2 \\
-\ell n\, 2 &< x < +\ell n\, 2.
\end{aligned}
$$

Thus three basic steps are required:

(1) Multiply X by the precomputed and stored constant $\log_2 e$ to identify I, f.

(2) Multiply f by the precomputed and stored constant $\log_e 2$ to generate x.

(3) Evaluate e^x via the normalization scheme discussed below.

Then,

$$e^X = e^x\, 2^I, \quad e^x \in [1/2,\, 2).$$

The first two of these operations may be performed by the multiplication algorithm of Section 4, and require no further discussion. It is the major purpose of this section to formulate an algorithm to evaluate e^x, $|x| < \ell n\, 2$. The algorithm proposed here is, in some sense, the dual of the algorithm proposed in Section 5 for $\ell n\, x$. Let

$$x = x - \ell n\Big(\prod_{i=0}^{M} e_i\Big) + \ell n\Big(\prod_{i=0}^{M} e_i\Big)$$

where

$$e_k = 1 + 1/2\, s_k 2^{-k}, \quad 1 \leq k \leq M.$$

Then,

$$x = \Big(x - \sum_{i=0}^{M} \ell n\, e_i\Big) + \ell n\Big(\prod_{i=0}^{M} e_i\Big)$$

$$e^x = e^{\left(x - \sum_{i=0}^{M} \ell n\, e_i\right)}\, e^{\ell n\left(\prod_{i=0}^{M} e_i\right)}$$

$$= e^{\left(x - \sum_{i=0}^{M} \ell n\, e_i\right)} \cdot \Big(\prod_{i=0}^{M} e_i\Big).$$

The set of multiplier constants is chosen such that

$$x + \sum_{i=0}^{M} (-\ell n\, e_i) \cong 0$$

so that,

$$e^x \cong \prod_{i=0}^{M} e_i .$$

The required set of precomputed constants, $\{-\ln e_i\}$ is exactly the same set required by the algorithm for $\ln x$.

Thus, to evaluate e^x, two simple recursions are performed.

$$x_{k+1} = x_0 + \sum_{i=0}^{k} (-\ln e_i) \qquad x_0 = x$$

$$= x_k + (-\ln e_k) \tag{8-1}$$

$$E_{k+1} = \prod_{i=0}^{k} e_i \qquad E_0 = 1$$

$$= E_k + s_k E_k 2^{-(k+1)} \tag{8-2}$$

where

$$s_k = \begin{cases} \bar{1} & \text{if} \quad x_k < -3/8 \cdot 2^{-k} \\ 0 & \text{otherwise} \\ 1 & \text{if} \quad x_k \geq +3/8 \cdot 2^{-k}. \end{cases}$$

8.2 Choice of initialization step

The initialization of this algorithm consists of a very small (five value) table-lookup and requires storage of four additional precomputed constants, namely, $e^{\pm 1/2}$, $e^{\pm 1/4}$.

TABLE 12

x	e_0	$\ell n\ e_0$
$[\frac{1}{2}, \ell n2)$	$e^{1/2}$	$\frac{1}{2}$
$[\frac{1}{4}, \frac{1}{2})$	$e^{1/4}$	$\frac{1}{4}$
$[-\frac{1}{4}, \frac{1}{4})$	1	0
$[-\frac{1}{2}, -\frac{1}{4})$	$e^{-1/4}$	$-\frac{1}{4}$
$(-\ell n2, -\frac{1}{2})$	$e^{-1/2}$	$-\frac{1}{2}$

Since $E_0 = 1$, no multiplication is required to form $E_1 = e_0$. Since $x_1 = x_0 - \ell n\ e_0$ and $\ell n\ e_0$ contains at most one non-zero bit, x_1 can also be formed quite easily.

It is shown in the next section that such an initialization leads to a convergent algorithm.

An example of the evaluation of e^x is given below.

<u>Example</u>: If $x = 0.6$, then $e^x = 1.82211880039051$. The error bound derived in Section 8.3 indicates that, for $M = 40$, the error should be less than 0.69×10^{-12}. The actual error in the approximation produced by the algorithm is less than 0.62×10^{-12}.

TABLE 13

k	s_k	x_{k+1}	E_{k+1}
1	0	0.10000000000000	1.64872127070013
2	1	-0.01778303565638	1.85481142953764
3	0	-0.01778303565638	1.85481142953764
4	0	-0.01778303565638	1.85481142953764
5	-1	-0.00203467868824	1.82583000095111
6	0	-0.00203467868824	1.82583000095111
7	0	-0.00203467868824	1.82583000095111
8	-1	-0.00007964385244	1.82226392673051
9	0	-0.00007964385244	1.82226392673051
10	0	-0.00007964385244	1.82226392673051
11	0	-0.00007964385244	1.82226392673051
12	0	-0.00007964385244	1.82226392673051
13	-1	-0.00001860683347	1.82215270456701
14	0	-0.00001860683347	1.82215270456701
15	-1	-0.00000334792799	1.82212490072326
16	0	-0.00000334792799	1.82212490972326
17	-1	0.00000046677655	1.82211794986838
18	0	0.00000046677655	1.82211794986838
19	0	0.00000046677655	1.82211794986838
20	1	-0.00000001006049	1.82211881872192
21	0	-0.00000001006049	1.82211881872192
22	0	-0.00000001006049	1.82211881872192
23	0	-0.00000001006049	1.82211881872192
24	0	-0.00000001006049	1.82211881872192
25	0	-0.00000001006049	1.82211881872192
26	-1	-0.00000000260991	1.82211880514608
27	0	-0.00000000260991	1.82211880514608
28	-1	-0.00000000074727	1.82211880175212
29	-1	0.00000000018405	1.82211880005514
30	0	0.00000000018405	1.82211880005514

(Continued)

TABLE 13 (Continued)

k	s_k	x_{k+1}	E_{k+1}
31	1	-0.0000000004878	1.82211880047938
32	0	-0.0000000004878	1.82211880047938
33	-1	0.0000000000943	1.82211880037332
34	0	0.0000000000943	1.82211880037332
35	0	0.0000000000943	1.82211880037332
36	1	0.0000000000216	1.82211880038658
37	0	0.0000000000216	1.82211880038658
38	1	0.0000000000034	1.82211880038989
39	0	0.0000000000034	1.82211880038989
40	0	0.0000000000034	1.82211880038989

8.3 Error bound

The error in the approximation

$$e^x \cong \prod_{i=0}^{M} e_i$$

is strongly related to the value of the function itself:

$$|e^x - E_{M+1}| = |e^{x_{M+1}} - 1| E_{M+1}$$

where

$$E_{M+1} = \prod_{i=0}^{M} e_i .$$

Thus, a bound on E_{M+1} is required in addition to a bound on x_{M+1}. In producing a bound on x_{M+1}, two preliminary results are helpful. First recall the power series

$$\ell n(1 + \delta) = \delta - 1/2\ \delta^2 + 1/3\ \delta^3 - 1/4\ \delta^4 + \ldots$$

$$[-1 < \delta \leq 1].$$

Then, for k = 1, 2, ...

$$|\ell n(1 + 2^{-k})| < |\ell n(1 - 2^{-k})|. \qquad (8\text{-}3)$$

Next it is shown by induction that

$$|\ell n(1 - 2^{-k})| < 3/2 \cdot 2^{-k}, \quad k = 1, 2, \ldots. \qquad (8\text{-}4)$$

Observe that since $\ell n(1 - 2^{-k}) < 0$, statement (8-4) is equivalent to

$$-\ell n(1 - 2^{-k}) < 3/2 \cdot 2^{-k}, \quad k = 1, 2, \ldots$$

or

$$\ell n(1 - 2^{-k}) > -3/2 \cdot 2^{-k}, \quad k = 1, 2, \ldots$$

or, exponentiating,

$$1 - 2^{-k} > e^{-3/2 \cdot 2^{-k}}, \quad k = 1, 2, \ldots \qquad (8\text{-}5)$$

which is in more convenient form for an inductive proof than is (8-4).

Proof: Since $e^{-3/4} \cong 0.47 < 1/2$, (8-5) is true for k = 1. For some k,

$$1 - 2^{-k} > e^{-3/2 \cdot 2^{-k}}.$$

Then surely,

$$1 - 2^{-k} + 1/4 \cdot 2^{-2k} > e^{-3/2 \cdot 2^{-k}}$$

or

$$(1 - 2^{-(k+1)})^2 > e^{-3/2 \cdot 2^{-k}}$$

or, taking positive roots,

$$1 - 2^{-(k+1)} > e^{-3/2 \cdot 2^{-(k+1)}}.$$

Q.E.D.

Now let us produce a bound on x_{M+1}. For the initialization chosen in the last section, one may easily show by direct computation that

$$x_1 = x_0 - \ell n\, e_0 \in [-1/4, +1/4).$$

<u>Hypothesis</u>: $|x_k| \leq 3/8 \cdot 2^{-(k-1)}$ for all k. The hypothesis is true for $k = 0$ since $|x_0| < \ell n\, 2 < 3/4$ and true for $k = 1$ since $|x_1| \leq 1/4 < 3/8$. The induction proof is completed by considering the mapping from the k^{th} step to the $(k+1)^{st}$ step.

<u>Range 1</u>: Suppose $-3/4 \cdot 2^{-k} \leq x_k < -3/8 \cdot 2^{-k}$ so that $s_k = \bar{1}$; then,

$$x_{k+1} = x_k - \ell n(1 - 2^{-(k+1)})$$

or,

$$-3/4 \cdot 2^{-k} + |\ell n(1-2^{-(k+1)})| \leq x_{k+1} < -3/8 \cdot 2^{-k} + |\ell n(1-2^{-(k+1)})|.$$

By (8-4),

$$|\ell n(1 - 2^{-(k+1)})| < 3/4 \cdot 2^{-k},$$

and by a power series expansion,

$$|\ell n(1 - 2^{-(k+1)})| = 2^{-(k+1)} + 1/2 \cdot 2^{-2(k+1)} + \ldots$$

or

$$|\ell n(1 - 2^{-(k+1)})| > 3/8 \cdot 2^{-k}.$$

Thus, for Range 1,

$$|x_{k+1}| \leq 3/8 \cdot 2^{-k}.$$

Range 2: Suppose $-3/8 \cdot 2^{-k} \leq x_k < 3/8 \cdot 2^{-k}$ so that $s_k = 0$; then $|x_{k+1}| = |x_k| \leq 3/8 \cdot 2^{-k}$.

Range 3: Suppose $3/8 \cdot 2^{-k} \leq x_k \leq 3/4 \cdot 2^{-k}$ so that $s_k = 1$; then

$$x_{k+1} = x_k - \ell n(1 + 2^{-(k+1)}).$$

From (8-3) and (8-4),

$$\ell n(1 + 2^{-(k+1)}) < 3/4 \cdot 2^{-k}.$$

But from the power series expansion,

$$\ell n(1 + 2^{-(k+1)} > 2^{-(k+1)} - 1/2 \cdot 2^{-2(k+1)}$$

or

$$\ell n(1 + 2^{-(k+1)}) > 1/2 \cdot 2^{-k} - 1/8 \cdot 2^{-k} \cdot 2^{-k}$$

which for $k \geq 1$ yields

$$\ell n(1 + 2^{-(k+1)}) > 3/8 \cdot 2^{-k}.$$

Thus,

$$3/8 \cdot 2^{-k} < \ell n(1 + 2^{-(k+1)}) < 3/4 \cdot 2^{-k}.$$

Therefore,

$$|x_{k+1}| \leq 3/8 \cdot 2^{-k}$$

for this range of x_k also.

Hence, for $k = 0, 1, 2, \ldots,$

$$|x_{k+1}| \leq 3/8 \cdot 2^{-k}.$$

Q.E.D.

Note further that

$$|e^{x_{M+1}} - 1| < |x_{M+1}| \, e^{|x_{M+1}|}$$

so that the error in the approximation to e^x is less than

$$|x_{M+1}| \, e^{|x_{M+1}|} \left(\prod_{i=0}^{M} e_i\right)$$

where

$$|x_{M+1}| \leq 3/8 \cdot 2^{-M}.$$

For any reasonable register length, say $M \geq 10$,

$$e^{|x_{M+1}|} < 1.001,$$

so the error may be bounded by

$$|e^x - \left(\prod_{i=0}^{M} e_i\right)| < 3/8 \cdot 2^{-M}(1.001) \left(\prod_{i=0}^{M} e_i\right)$$

or

$$|e^x - E_{M+1}| < 0.376 \cdot 2^{-M} \, E_{M+1}.$$

By direct computation,

$$\prod_{i=0}^{M} e_i \leq e^{1/2} \prod_{i=1}^{M} (1 + 1/2 \; s_i 2^{-i})$$

$$< e^{1/2} \prod_{i=1}^{M} (1 + 2^{-(i+1)})$$

$$< 3.31.$$

Thus surely

$$|e^x - E_{M+1}| < 1.25 \cdot 2^{-M}.$$

But since E_{M+1} is a close approximation to e^x, it is substantially less than 3.31:

$$E_{M+1} \leq \frac{e^x}{1 - 0.376 \cdot 2^{-M}} \leq \frac{2}{1 - 0.376 \cdot 2^{-M}}$$

$$E_{M+1} < 2(1 + 1/2 \cdot 2^{-M}).$$

Therefore,

$$|e^x - E_{M+1}| < 0.376 \cdot 2^{-M} \cdot 2(1 + 1/2 \cdot 2^{-M}) < 0.753 \cdot 2^{-M}$$

for $M \geq 10$.

Hence, the performance of $M + 1$ steps beyond the initialization suffices to guarantee M correct bits in the approximation to e^x.

8.4 Experimental estimate of speed

The Monte Carlo estimate of the mean probability of a zero is 0.669, with a corresponding shift average of 3.04.

8.5 Implementation

Making the usual transformation, $u_k = 2^k x_k$, such that $|u_k| < 1$ for all k, and rewriting recursion (8-1), one obtains,

$$u_{k+1} = 2\left[u_k + 2^k\left(-\ell n(1 + s_k 2^{-(k+1)})\right)\right], \quad u_0 = x \qquad (8\text{-}6)$$

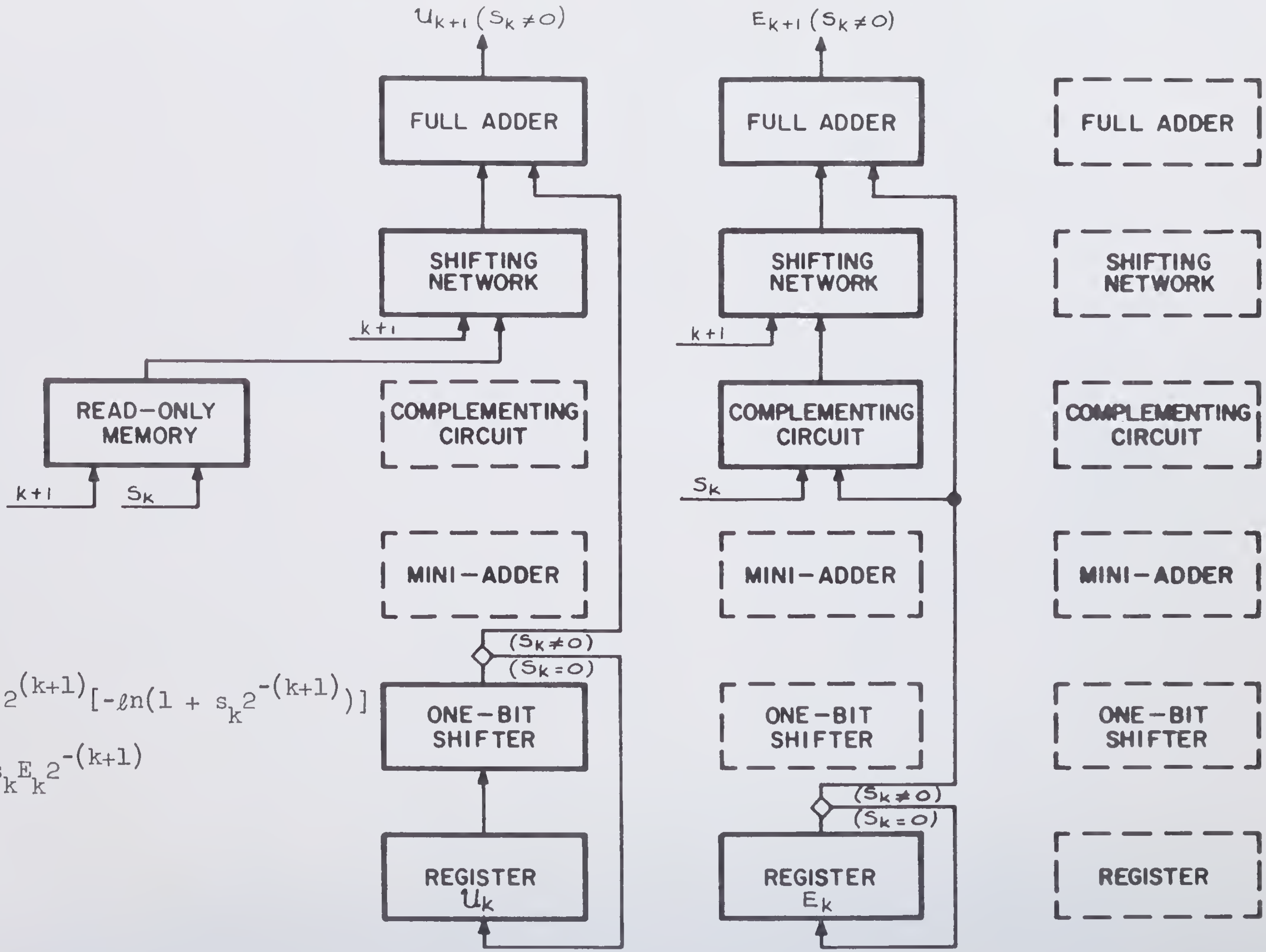
with recursion (8-2) remaining unchanged.

$$E_{k+1} = E_k + s_k E_k 2^{-(k+1)}, \quad E_0 = 1, \quad E_1 = e_0. \tag{8-7}$$

A hardware configuration to implement these recursions is shown in Figure 8.

8.6 Concluding remark

An algorithm for evaluating e^X in three multiplication cycle times has been proposed. The algorithm requires storage of only a few precomputed constants not required by other algorithms; the hardware required for implementation fits within the requirements of other algorithms previously proposed.



$$u_{k+1} = 2u_k + 2^{(k+1)}[-\ell n(1 + s_k 2^{-(k+1)})]$$

$$E_{k+1} = E_k + s_k E_k 2^{-(k+1)}$$

FIGURE 8. Block diagram for exponential

9. THE ALGORITHM FOR TANGENT (OR COTANGENT)

9.1 Basic algorithm

The tangent (or cotangent) of any argument X, $0 \leq X < 2\pi$, may be formed in two multiplication cycle times beyond an initial range reduction. From the standard identity,

$$\tan(X + \pi) = \tan X$$

it is clear that one need only consider arguments in the range $0 \leq X < \pi$. Further, from the identity,

$$\tan(X + \pi/2) = -\operatorname{ctn} X$$

it is clear that the range may be reduced to $0 \leq X < \pi/2$. Hence, the initial range reduction requires storage of the values π, $\pi/2$; whether one stores both values explicitly or obtains one as a shifted version of the other is a moot question.

The algorithm proposed here to evaluate tan X (or ctn X), $0 \leq X < \pi/2$, requires the use of the complex exponential,

$$e^{jX} = \cos X + j \sin X$$

where the operator j represents a 90° counterclockwise (positive) rotation in the complex plane. The normalization procedure is conceptually identical to that employed in the division algorithm, except that the multiplier constants are complex numbers. Let

$$e^{jX} = e^{jX} \frac{\prod_{i=0}^{M} t_i}{\prod_{i=0}^{M} t_i}$$

where

$$t_k = |t_k| e^{j\varphi_k}, \quad |t_k| > 0.$$

Then,

$$\prod_{i=0}^{M} t_i = \left(\prod_{i=0}^{M} |t_i| \right) e^{j \sum_{i=0}^{M} \varphi_i}$$

and

$$e^{jX} = e^{j\left(X - \sum_{i=0}^{M} \varphi_i\right)} \frac{\prod_{i=0}^{M} t_i}{\prod_{i=0}^{M} |t_i|} .$$

If the multipliers are chosen such that

$$X - \sum_{i=0}^{M} \varphi_i \cong 0$$

then

$$e^{jX} = \cos X + j \sin X \cong K \prod_{i=0}^{M} t_i$$

where

$$K = \frac{1}{\prod_{i=0}^{M} |t_i|} .$$

Hence, if

$$T_{M+1} = R_{M+1} + j\, I_{M+1} = \prod_{i=0}^{M} t_i$$

with R_{M+1}, I_{M+1} real, then

$$\tan X \cong \frac{I_{M+1}}{R_{M+1}}$$

$$\text{ctn } X \cong \frac{R_{M+1}}{I_{M+1}}$$

independent of K.

For simplicity sake, assume that the tangent is required. The range of the operand may be further limited to $0 \leq X < \pi/4$ as follows:

(1) If $0 \leq X < \pi/4$, let $x = X$ and

$$\tan X = \tan x \cong \frac{I_{M+1}}{R_{M+1}} ;$$

(2) If $\pi/4 \leq X < \pi/2$, let $x = \pi/2 - X$ and

$$\tan X = \text{ctn } x \cong \frac{R_{M+1}}{I_{M+1}} .$$

Having performed the indicated range reductions, one need only formulate an algorithm to compute the real and imaginary parts of e^{jx}, a final division being required to evaluate the tangent.

Four constraints are placed on the set of multipliers: (1) the summation of the angles must approach x; (2) the magnitudes must be non-zero; (3) the continued product of the multipliers must be easy to compute in rectangular coordinates; (4) approximately two-thirds of the multipliers, on the average, should be 1 + j0, if possible. The following choice for the k^{th} multiplier satisfies these constraints:

$$t_k = 1 + j\ 1/2\ s_k 2^{-k}$$

where

$$s_k = \begin{cases} \bar{1} & \text{if} \quad x_k < -3/8 \cdot 2^{-k} \\ 0 & \quad\ \text{otherwise} \\ 1 & \text{if} \quad x_k \geq 3/8 \cdot 2^{-k} \end{cases}$$

$$x_{k+1} = x_0 - \sum_{i=0}^{k} \varphi_i, \qquad x_0 = x.$$

Observe that one need not compute

$$K = \frac{1}{\prod_{i=0}^{M} |t_i|}$$

since it is merely a scale factor which disappears during the final division.

With the exceptions of a few additions required for the initial range reductions and the final division, one need only perform the recursions developed below.

$$\begin{aligned} x_{k+1} &= x_0 - \sum_{i=0}^{k} \varphi_i \qquad\qquad x_0 = x \\ &= x_k - \varphi_k \\ &= x_k - \tan^{-1}(s_k 2^{-(k+1)}) \\ &= x_k - s_k \tan^{-1}(2^{-(k+1)}) \end{aligned} \tag{9-1}$$

$$\begin{aligned} T_{k+1} &= R_{k+1} + j\, I_{k+1} \qquad\qquad T_0 = 1 + j0 \\ &= \prod_{i=0}^{k} t_i \\ &= (R_k + j\, I_k)(1 + j\, s_k 2^{-(k+1)}) \\ &= (R_k - s_k 2^{-(k+1)} I_k) + j(I_k + s_k 2^{-(k+1)} R_k) \end{aligned}$$

i.e.,

$$R_{k+1} = R_k - s_k I_k 2^{-(k+1)} \tag{9-2}$$

$$I_{k+1} = I_k + s_k R_k 2^{-(k+1)}. \tag{9-3}$$

The set of constants, $\{\tan^{-1}(2^{-i})\}$, must be precomputed and stored in the ROM. A series expansion

$$\tan^{-1}\delta = \delta - 1/3\,\delta^3 + 1/5\,\delta^5 - 1/7\,\delta^7 + \ldots \qquad [|\delta| < 1]$$

indicates that only one-third of these constants need be explicitly stored since, for $k \geq [M/3]$, $\tan^{-1}(2^{-k}) = 2^{-k}$ to machine accuracy.

9.2 Choice of initialization step

At the cost of storing one additional precomputed constant, one may choose

$$t_0 = 1 + j\tan(\pi/8)$$

so that

$$x_1 = x_o - \pi/8 \quad \epsilon\ [-\pi/8, +\pi/8)$$

$$R_1 = 1$$

$$I_1 = \tan(\pi/8),$$

the value $\tan(\pi/8)$ being stored.

An example of the normalization procedure and the step-by-step values of R_k and I_k is listed in Table 14. Presumably the division algorithm is used to compute the ratio I_{M+1}/R_{M+1} to produce the tangent.

Example: If $x = 0.6$, then $\tan x = 0.68413680834169$. The algorithm produces $R_{M+1} = 0.92130871026429$, $I_{M+1} = 0.63030120053774$. The ratio is 0.68413680834183, which differs from the correct value by 0.14×10^{-12}. This error is within the error bound of 0.68×10^{-12}.

TABLE 14

k	s_k	x_{k+1}	R_{k+1}	I_{k+1}	I_{k+1}/R_{k+1}
1	1	-0.03767774482559	0.89644660940673	0.66421356237310	0.74094045914533
2	0	-0.03767774482559	0.89644660940673	0.66421356237310	0.74094045914533
3	0	-0.03767774482559	0.89644660940673	0.66421356237310	0.74094045914533
4	-1	-0.00643791139532	0.91720328323089	0.63619960582913	0.69362988277593
5	0	-0.00643791139532	0.91720328323089	0.63619960582913	0.69362988277593
6	-1	0.00137442966478	0.92217359265143	0.62903395517889	0.68212098046562
7	0	0.00137442966478	0.92217359265143	0.62903395517889	0.68212098046562
8	0	0.00137442966478	0.92217359265143	0.62903395517889	0.68212098046562
9	1	0.00039786747522	0.92155930167957	0.62993451532797	0.68355288062297
10	1	-0.00009041373597	0.92125171646701	0.63038449545574	0.68426954782050
11	0	-0.00009041373597	0.92125171646701	0.63038449545574	0.68426954782050
12	0	-0.00009041373597	0.92125171646701	0.63038449545574	0.68426954782050
13	-1	-0.00002937857980	0.92129019208319	0.63032826671328	0.68417993823206
14	-1	0.00000113899832	0.92130942817531	0.63030015116787	0.68413513624429
15	0	0.00000113899832	0.92130942817531	0.63030015116787	0.68413513624429
16	0	0.00000113899832	0.92130942817531	0.63030015116787	0.68413513624429
17	0	0.00000113899832	0.92130942817531	0.63030015116787	0.68413513624429
18	0	0.00000113899832	0.92130942817531	0.63030015116787	0.68413513624429
19	1	0.00000018532400	0.92130882707424	0.63030102979701	0.68413653627809
20	0	0.00000018532400	0.92130882707424	0.63030102979701	0.68413653627809

(Continued)

TABLE 14 (Continued)

k	s_k	x_{k+1}	R_{k+1}	I_{k+1}	I_{k+1}/R_{k+1}
21	1	-0.0000005309458	0.92130867679877	0.63030124945419	0.68413688628683
22	0	-0.0000005309458	0.92130867679877	0.63030124945419	0.68413688628683
23	-1	0.0000000651007	0.92130871436765	0.63030119453987	0.68413679878463
24	0	0.0000000651007	0.92130871436765	0.63030119453987	0.68413679878463
25	0	0.0000000651007	0.92130871436765	0.63030119453987	0.68413679878463
26	1	-0.0000000094051	0.92130870967154	0.63030120140416	0.68413680972241
27	0	-0.0000000094051	0.92130870967154	0.63030120140416	0.68413680972241
28	0	-0.0000000094051	0.92130870967154	0.63030120140416	0.68413680972241
29	-1	-0.0000000000919	0.92130871025855	0.63030120054612	0.68413680835519
30	0	-0.0000000000919	0.92130871025855	0.63030120054612	0.68413680835519
31	0	-0.0000000000919	0.92130871025855	0.63030120054612	0.68413680835519
32	0	-0.0000000000919	0.92130871025855	0.63030120054612	0.68413680835519
33	0	-0.0000000000919	0.92130871025855	0.63030120054612	0.68413680835519
34	0	-0.0000000000919	0.92130871025855	0.63030120054612	0.68413680835519
35	0	-0.0000000000919	0.92130871025855	0.63030130054612	0.68413680835519
36	-1	-0.0000000000192	0.92130871026314	0.63030120053918	0.68413680834451
37	0	-0.0000000000192	0.92130871026314	0.63030120053918	0.68413680834451
38	-1	-0.0000000000010	0.92130871264287	0.63030120053774	0.68413680834183
39	0	-0.0000000000010	0.92130871264287	0.63030120053774	0.68413680834183
40	0	-0.0000000000010	0.92130871264287	0.63030120053774	0.68413680834183

9.3 Error bound

It is first shown that, for $k \geq 2$, $|x_k| \leq 3/4 \cdot 2^{-k}$, and that if $u_k = 2^k x_k$ then $|u_k| < 1$ for all k.

Recall that the range reductions resulted in $|x_0| < \pi/4$ so that $|u_0| < 1$. Also, since $|x_1| \leq \pi/8$, $|u_1| < 1$. Consider the choice of t_1:

$$t_1 = 1 + 1/4\ s_1$$

$$\varphi_1 = s_1 \tan^{-1}(1/4)$$

$$\cong 0.24498\ s_1$$

where

$$s_1 = \begin{cases} \bar{1} & \text{if} \quad x_1 < -3/16 \\ 0 & \quad\quad \text{otherwise} \\ 1 & \text{if} \quad x_1 \geq 3/16. \end{cases}$$

Range 1: Suppose $-\pi/8 \leq x_1 < -3/16$ so that $s_1 = \bar{1}$. Then

$$x_2 = x_1 - \varphi_1$$

$$= x_1 + \tan^{-1}(1/4)$$

so that

$$|x_2| < 0.16 < 3/16$$

$$|u_2| < 0.64 < 1.$$

Range 2: Suppose $-3/16 \leq x_1 < 3/16$ so that $s_1 = 0$; then $|x_2| = |x_1| \leq 3/16$ and $|u_2| < 1$.

Range 3: Suppose $3/16 \leq x_1 < \pi/8$ so that $s_1 = 1$. By symmetry, $|x_2| < 3/16$, $|u_2| < 1$.

<u>Hypothesis</u>: For $k \geq 2$, $|x_k| \leq 3/4 \cdot 2^{-k}$, $|u_k| < 1$.

<u>Proof</u>: The hypothesis has been shown to be correct for $k = 2$. The proof is completed by induction.

<u>Range 1</u>: Suppose $-3/4 \cdot 2^{-k} \leq x_k < -3/8 \cdot 2^{-k}$. Then,

$$x_{k+1} = x_k + \tan^{-1}(2^{-(k+1)})$$

and

$$-3/4 \cdot 2^{-k} + \tan^{-1}(2^{-(k+1)}) \leq x_{k+1} < -3/8 \cdot 2^{-k} + \tan^{-1}(2^{-(k+1)}).$$

Now,

$$\tan^{-1}(2^{-(k+1)}) = 2^{-(k+1)} - 1/3 \cdot 2^{-3(k+1)} + 1/5 \cdot 2^{-5(k+1)} - \ldots$$

so

$$2^{-(k+1)} - 1/3 \cdot 2^{-3(k+1)} < \tan^{-1}(2^{-(k+1)}) < 1/2 \cdot 2^{-k}.$$

Since $k \geq 2$,

$$2^{-k}(1/2 - 1/1536) < \tan^{-1}(2^{-(k+1)}) < 1/2 \cdot 2^{-k}$$

or

$$767/1536 \cdot 2^{-k} < \tan^{-1}(2^{-(k+1)}) < 1/2 \cdot 2^{-k}.$$

Thus,

$$2^{-k}(-3/4 + 767/1536) < x_{k+1} < 2^{-k}(-3/8 + 1/2)$$

or

$$-385/1536 \cdot 2^{-k} < x_{k+1} < 1/8 \cdot 2^{-k}.$$

Hence,

$$|x_{k+1}| < 3/4 \cdot 2^{-(k+1)}$$

here.

Range 2: Suppose $-3/8 \cdot 2^{-k} \leq x_k < +3/8 \cdot 2^{-k}$. Then $s_k = 0$ and $|x_{k+1}| \leq 3/4 \cdot 2^{-(k+1)}$.

Range 3: The proof for this range follows that above by symmetry considerations. Hence, $|x_k| \leq 3/4 \cdot 2^{-k}$; also $|u_k| < 1$. Finally, $|x_{M+1}| \leq 3/8 \cdot 2^{-M}$.

While the error in the normalization process has been bounded by $3/8 \cdot 2^{-M}$, it is not clear that the values of cos x and sin x are known to such precision. An approximate error bound on the error in tan x is developed below.

First observe that

$$e^{jx} = K\, e^{jx_{M+1}}(R_{M+1} + j\, I_{M+1})$$

so that, equating real and imaginary parts,

$$K\, R_{M+1} = \cos(x - x_{M+1})$$

$$K\, I_{M+1} = \sin(x - x_{M+1}),$$

where $K\, R_{M+1}$ is the approximation to cos x and $K\, I_{M+1}$ is the approximation to sin x. The error in the approximation to tan x is thus given by

$$E_{\tan x} = \tan x - \tan\,(x - x_{M+1})$$

$$= \tan x - \frac{\tan x - \tan x_{M+1}}{1 + \tan x \tan x_{M+1}}$$

$$= \frac{\tan x_{M+1} \sec^2 x}{1 + \tan x \tan x_{M+1}} .$$

Since $0 \leq x < \pi/4$, $1 \leq \sec^2 x < 2$ and $0 \leq \tan x < 1$. Therefore,

$$|E_{\tan x}| < \frac{2 \tan |x_{M+1}|}{1 - \tan|x_{M+1}|} .$$

But $|x_{M+1}| \leq 3/8 \cdot 2^{-M}$, so, to first order, $|\tan x_{M+1}| < 3/8 \cdot 2^{-M}$ and

$$|E_{\tan x}| \leq 3/4 \cdot 2^{-M}.$$

When the value of X exceeds $\pi/4$, $x = \pi/2 - X$, and one computes ctn x rather than tan x. The analogous error bound is given by

$$E_{\text{ctn } x} = \text{ctn } x + \frac{1 + \text{ctn } x \text{ ctn } x_{M+1}}{\text{ctn } x - \text{ctn } x_{M+1}}$$

$$= \frac{\csc^2 x}{\text{ctn } x - \text{ctn } x_{M+1}}$$

$$= \frac{2 \tan x_{M+1} \csc 2x}{\tan x_{M+1} - \tan x}.$$

Therefore, to first order,

$$|E_{\text{ctn } x}| \cong \frac{2|x_{M+1}|\csc 2x}{|\tan x - x_{M+1}|} .$$

It may be observed that as x approaches zero (X approaches $\pi/2$) the error increases without bound; such behavior is to be expected since tan X itself increases without bound. If $x \gg |x_{M+1}|$, then the error may be approximated by

$$|E_{\text{ctn } x}| \cong |x_{M+1}| \csc^2 x, \quad x \gg |x_{M+1}|$$

which is seen to approach $3/4 \cdot 2^{-M}$ as x approaches $\pi/4$, matching the error bound for the tangent.

If one wishes to achieve a small error in the case where $X \to \pi/2$, $(x \to 0)$, it may be best to turn to a power series expansion such as

$$\operatorname{ctn} x = 1/x - 1/3\, x - 1/45\, x^3 - 2/945\, x^5 - \ldots$$

$$- \frac{(-1)^{n-1} 2^{2n} B_{2n}}{(2n)!} x^{2n-1} - \ldots$$

$$[|x| < \pi]$$

where B_n is a Bernoulli number. When x is very small, the first few terms in such a series suffice to yield an acceptably small error. For example, if $x < \sqrt{1/3 \cdot 2^{-(M+1)}}$, then $\operatorname{ctn} x = 1/x$ to machine accuracy.

Similarly, if the tangent of X is required and X is sufficiently small, it would be better to use a power series expansion here also.

$$\tan X = X + 1/3\, X^3 + 2/15\, X^5 + \ldots.$$

$$[|X| < \pi/2]$$

If $X < \sqrt{3 \cdot 2^{-(M+1)}}$, then $\tan X = X$ to machine accuracy.

9.4 Experimental estimate of speed

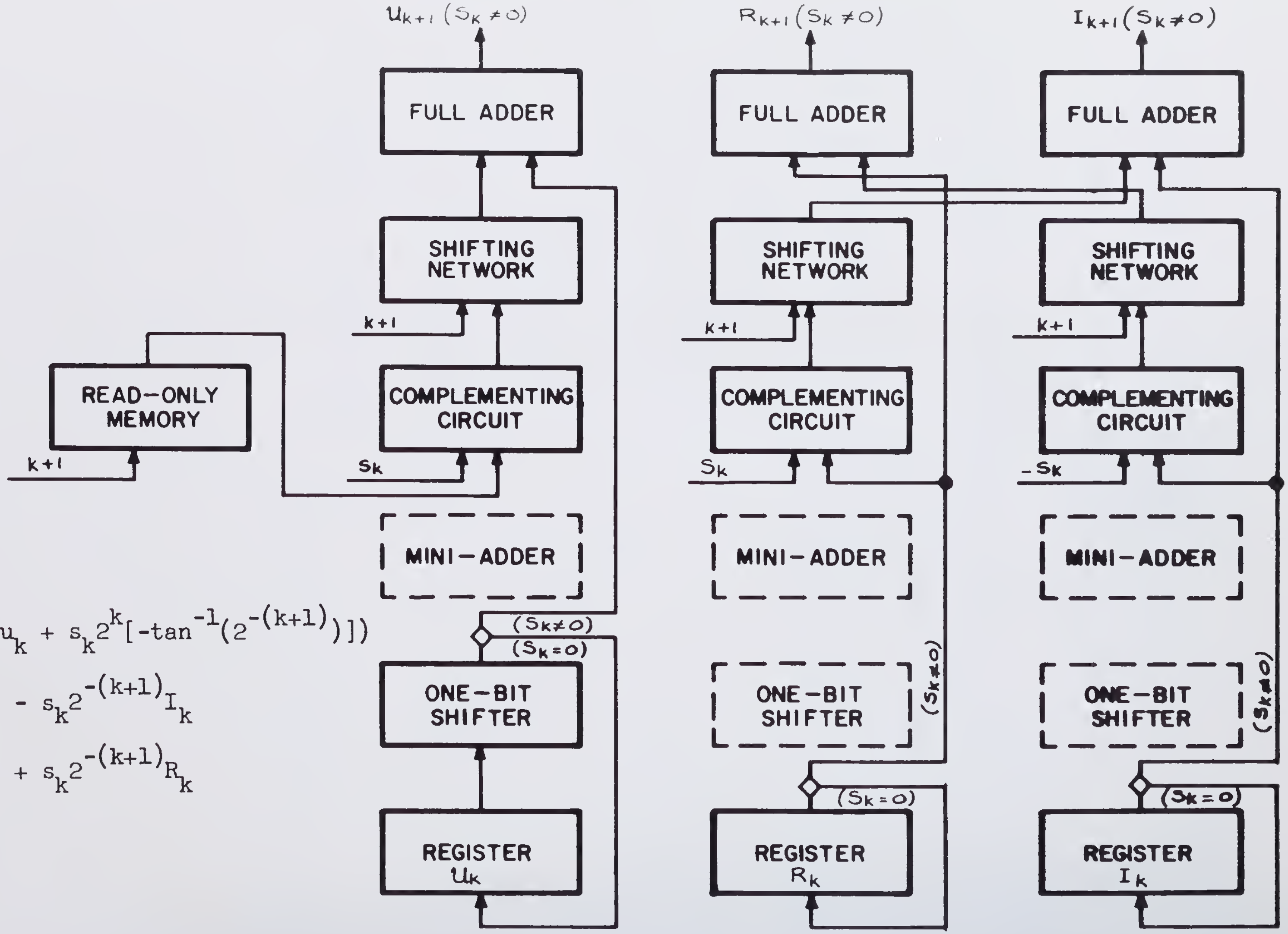
The normalization of the operand x and the parallel evaluation of the approximations to cosine and sine require approximately one multiplication cycle time. The Monte Carlo estimate of the mean probability of a zero for this process is 0.653, with a corresponding shift average of 2.90. In addition, as many as three range reduction subtractions are required to obtain x. A final division to evaluate either tan x or ctn x is also required. Hence, an average of slightly more than two multiplication cycle times is required to evaluate tan X.

9.5 Implementation

Figure 9 shows a hardware configuration to implement recursion relations (9-1), (rewritten in terms of $u_k = 2^k x_k$), (9-2), and (9-3).

9.6 Concluding remark

An algorithm for computing tan X (or ctn X) in approximately two multiplication cycle times has been developed. It has been shown that, for certain values of X, it may be advisable to consider a power series expansion as an alternative to minimize the error.



$$u_{k+1} = 2(u_k + s_k 2^k[-\tan^{-1}(2^{-(k+1)})])$$

$$R_{k+1} = R_k - s_k 2^{-(k+1)} I_k$$

$$I_{k+1} = I_k + s_k 2^{-(k+1)} R_k$$

FIGURE 9. Block diagram for tangent

10. THE ALGORITHM FOR COSINE AND SINE

10.1 Basic algorithm

The algorithm for computing the cosine and sine of an argument X, $0 \leq X < 2\pi$, differs from the algorithm for tangent in only two respects. First, the constant K must be computed since the cosine and sine are explicitly required, rather than merely their ratio; second, the final division is unnecessary. The chief concern here is thus the evaluation of K.

Recall that

$$K = \frac{1}{\prod_{i=0}^{M} \sqrt{1 + s_i^2 \, 2^{-2(i+1)}}} \, .$$

Clearly, if the algorithm is performed non-redundantly with $s_k \in \{\bar{1}, 1\}$,* then K can be precomputed and stored. Such a choice, however, guarantees that at every step an addition must be performed, thus slowing down the algorithm considerably. A compromise between Specker's non-redundant algorithm and the fully redundant ($s_k \in \{\bar{1}, 0, 1\}$, $k = 1, 2, \ldots, M$) algorithm one would like to perform is developed here.

Consider an expansion of $1/|t_k|$:

$$\frac{1}{|t_k|} = \frac{1}{\sqrt{1 + s_k^2 \, 2^{-2(k+1)}}}$$

$$= 1 - s_k^2 \, 2^{-(2k+3)} + s_k^2 \, 2^{-(4k+6)}$$

$$+ s_k^2 \, 2^{-(4k+7)} + \text{higher order terms.}$$

---

* Such an algorithm was proposed by Specker.[1]

For $k \leq (\frac{M-6}{4})$, at least three terms in the expansion are required to represent $1/|t_k|$ to machine accuracy; it is preferable to disallow $s_k = 0$ and precompute the constant required. However, for $k > (\frac{M-6}{4})$, it is preferable to allow $s_k = 0$; the computation requires a simple correction factor. For $(\frac{M-6}{4}) < k \leq (\frac{M-3}{2})$,

$$\frac{1}{|t_k|} = 1 - s_k^2 \, 2^{-(2k+3)}$$

to machine accuracy. For $k > (\frac{M-3}{2})$,

$$\frac{1}{|t_k|} = 1$$

to machine accuracy and no correction factor is necessary.

In summary, then, the following selection rules are proposed:

$$t_0{}^{*} = 1 + j \tan \pi/8$$

$$t_k = 1 + j\, 1/2\, s_k 2^{-k}, \quad k = 1, 2, \ldots, M$$

where

(1) for $k = 1, 2, \ldots, [\frac{M-6}{4}] + 1$,

---

* The constants

$$K' = \frac{1}{|t_0|} \prod_{i=0}^{[\frac{M-6}{4}]+1} (1 + 2^{-2(i+1)})^{-1/2}$$

$$K'' = \frac{\mathrm{Im}\{t_0\}}{|t_0|} \prod_{i=0}^{[\frac{M-6}{4}]+1} (1 + 2^{-2(i+1)})^{-1/2}$$

are precomputed and stored and serve as initial values R_1 and I_1, respectively.

$$s_k = \begin{cases} \bar{1} & \text{if} \quad x_k \le 0 \\ 1 & \text{if} \quad x_k > 0 \end{cases}$$

(One addition cycle time is required per step.)

(2) for $k = [\frac{M-6}{4}] + 2, \ldots, [\frac{M-3}{2}] + 1,$

$$s_k = \begin{cases} \bar{1} & \text{if} \quad x_k < -3/8 \cdot 2^{-k} \\ 0 & \text{otherwise} \\ 1 & \text{if} \quad x_k \ge 3/8 \cdot 2^{-k} \end{cases}$$

$$\frac{1}{|t_k|} = 1 - s_k^2 \, 2^{-(2k+3)}$$

(Two addition cycle times are required if $s_k \neq 0$, leading to an average of two-thirds addition cycle times per step.)

(3) for $k = [\frac{M-3}{2}] + 2, \ldots, M,$ s_k as above,

$$\frac{1}{|t_k|} = 1.$$

(A single addition cycle time is required if $s_k \neq 0$, leading to an average of one-third per step.)

Since roughly one-quarter of the steps are non-redundant, one-quarter are redundant but require two additive cycle times per non-zero step, and one-half are redundant and require one addition cycle time per non-zero step, approximately

$$1/4 \; M + 1/4 \cdot 2 \cdot M/3 + 1/2 \cdot M/3 = 7/12 \; M$$

addition cycle times, on the average, beyond initial range reduction, if any, are required to compute the cosine and sine.

Other choices for the set of multiplier constants $\{t_i\}$ were studied with the conclusion that the set chosen appears to be near an optimum. A brief discussion of the reasoning leading to this conclusion is presented in Appendix B. It may be recognized that the problem faced here is identical to that faced in attempting to introduce redundancy into Volder's[2] CORDIC vector rotation scheme.

10.2 Example

Having chosen

$$t_0 = 1 + j \tan(\pi/8),$$

so that $\varphi_0 = \pi/8$, $R_1 = K'$, $I_1 = K''$, it is possible to carry out an example.

Example: If x = 0.6, then cos x = 0.82533561490968 and sin x = 0.56464247339504. The algorithm produces approximations which are in error by 0.4×10^{-13} and 0.1×10^{-12}, respectively. The error bound, derived in Section 10.3, for both cosine and sine is 0.45×10^{-12}.

TABLE 15

k	s_k	x_{k+1}	R_{k+1}	I_{k+1}
0	0	0.20730091830128	0.88706417837978	0.36743401338025
1	1	-0.03767774482559	0.79520567503472	0.58920005797520
2	-1	0.08667724972117	0.86885568228162	0.48979934859586
3	1	0.02425843972522	0.83824322299437	0.54410282873846
4	1	-0.00698139370505	0.82124000959630	0.57029792945703
5	-1	0.00864233491542	0.83015091474406	0.55746605430709
6	1	0.00082999385532	0.82579571119479	0.56395160832853
7	1	-0.00307623627664	0.82359277522476	0.56717737282538
8	-1	-0.00112311376016	0.82470054353106	0.56556879318627

(Continued)

TABLE 15 (Continued)

k	s_k	x_{k+1}	R_{k+1}	I_{k+1}
9	-1	-0.00014655157061	0.82525285680565	0.56476342156173
10	0	-0.00014655157061	0.82525285680565	0.56476342156173
11	0	-0.00014655157061	0.82525285680565	0.56476342156173
12	-1	-0.00002448125871	0.82532179150388	0.56466267848054
13	0	-0.00002448125871	0.82532179150388	0.56466267848054
14	-1	0.00000603631940	0.82523902325696	0.56463749139536
15	0	0.00000603631940	0.82533902325696	0.56463749139536
16	1	-0.00000159307513	0.82533471539075	0.56464378821596
17	0	-0.00000159307513	0.82533471539075	0.56464378821596
18	-1	0.00000031427351	0.82533579236181	0.56464221401389
19	0	0.00000031427351	0.82533579236181	0.56464221401389
20	0	0.00000031427351	0.82533579236181	0.56464221401389
21	1	0.00000007585493	0.82533565774061	0.56464241078928
22	0	0.00000007585493	0.82533565774061	0.56464241078928
23	1	0.00000001625028	0.82533562408530	0.56464245998312
24	0	0.00000001625028	0.82533562408530	0.56464245998312
25	1	0.00000000134912	0.82533561567147	0.56464247228158
26	0	0.00000000134912	0.82533561567147	0.56464247228158
27	0	0.00000000134912	0.82533561567147	0.56464247228158
28	0	0.00000000134912	0.82533561567147	0.56464247228158
29	1	0.00000000041780	0.82533561514561	0.56464247305023
30	1	-0.00000000004786	0.82533561488268	0.56464247343456
31	0	-0.00000000004786	0.82533561488268	0.56464247343456
32	0	-0.00000000004786	0.82533561488268	0.56464247343456
33	-1	0.00000000001034	0.82533561491554	0.56464247338652
34	0	0.00000000001034	0.82533561491554	0.56464247338652
35	0	0.00000000001034	0.82533561491554	0.56464247338652
36	1	0.00000000000307	0.82533561491144	0.56464247339252
37	1	-0.00000000000057	0.82533561490938	0.56464247339552
38	0	-0.00000000000057	0.82533561490938	0.56464247339552
39	0	-0.00000000000057	0.82533561490938	0.56464247339552
40	-1	-0.00000000000011	0.82533561490964	0.56464247339515

10.3 Error bound

It is shown that the errors in the approximations to both cosine and sine are less than $2^{-(M+1)}$.

From the choice of t_0, $|x_1| \leq \pi/8 < \tan^{-1}(1/2)$. For $1 \leq k \leq [\frac{M-6}{4}] + 1$,

$$\varphi_k = s_k \tan^{-1}(2^{-(k+1)})$$

where

$$s_k = \begin{cases} \bar{1} & \text{if} \quad x_k \leq 0 \\ 1 & \text{if} \quad x_k > 0. \end{cases}$$

During these steps,

$$|x_k| \leq \tan^{-1}(2^{-k}).$$

Proof: For $k = 1$, $|x_1| \leq \tan^{-1}(1/2)$ as indicated above. The inductive proof that $|x_k| \leq \tan^{-1}(2^{-k})$ is based on the observation that

$$\tan^{-1}(2^{-k}) - \tan^{-1}(2^{-(k+1)}) < \tan^{-1}(2^{-(k+1)})$$

i.e.,

$$\tan^{-1}(2^{-k}) < 2 \tan^{-1}(1/2 \cdot 2^{-k}).$$

This observation follows from the power series

$$\tan^{-1} \delta = \delta - 1/3\ \delta^3 + 1/5\ \delta^5 - 1/7\ \delta^7 + \ldots \qquad [|\delta| < 1].$$

Since for some k,

$$|x_k| \leq \tan^{-1}(2^{-k})$$

and since, from the selection rules,

$$|x_{k+1}| \leq \tan^{-1}(2^{-k}) - \tan^{-1}(2^{-(k+1)}),$$

it follows from the above observation that

$$|x_{k+1}| \leq \tan^{-1}(2^{-(k+1)}).$$

Thus, if $t = [\frac{M-6}{4}] + 1$, at the end of this sequence of steps

$$|x_{t+1}| \leq \tan^{-1}(2^{-(t+1)}).$$

For $t + 1 \leq k \leq M$,

$$\varphi_k = s_k \tan^{-1}(2^{-(k+1)})$$

where

$$s_k = \begin{cases} \bar{1} & \text{if} \quad x_k < -3/8 \cdot 2^{-k} \\ 0 & \text{otherwise} \\ 1 & \text{if} \quad x_k \geq 3/8 \cdot 2^{-k}. \end{cases}$$

During these steps also,

$$|x_k| \leq \tan^{-1}(2^{-k}).$$

Proof: From above, $|x_{t+1}|$ satisfies the hypothesis. The induction proof is completed in the usual fashion.

Range 1: Suppose $-\tan^{-1}(2^{-k}) \leq x_k < -3/8 \cdot 2^{-k}$; then $x_{k+1} = x_k + \tan^{-1}(2^{-(k+1)})$ and

$$-\tan^{-1}(2^{-k}) + \tan^{-1}(2^{-(k+1)}) \leq x_{k+1} < -3/8 \cdot 2^{-k} + \tan^{-1}(2^{-(k+1)}).$$

Since

$$\tan^{-1}(2^{-k}) - \tan^{-1}(2^{-(k+1)}) < \tan^{-1}(2^{-(k+1)})$$

and

$$\tan^{-1}(2^{-k}) - 3/8 \cdot 2^{-k} < \tan^{-1}(2^{-(k+1)})$$

then

$$|x_{k+1}| < \tan^{-1}(2^{-(k+1)}).$$

Range 2: Suppose $-3/8 \cdot 2^{-k} \leq x_k < 3/8 \cdot 2^{-k}$; then

$$|x_{k+1}| \leq 3/8 \cdot 2^{-k} < \tan^{-1}(2^{-(k+1)}).$$

Range 3: Suppose $3/8 \cdot 2^{-k} \leq x_k < \tan^{-1}(2^{-k})$. The argument for this range follows, by symmetry, from that for Range 1.

Q.E.D.

The approximation

$$e^{j\,x_{M+1}} \cong 1 + j0$$

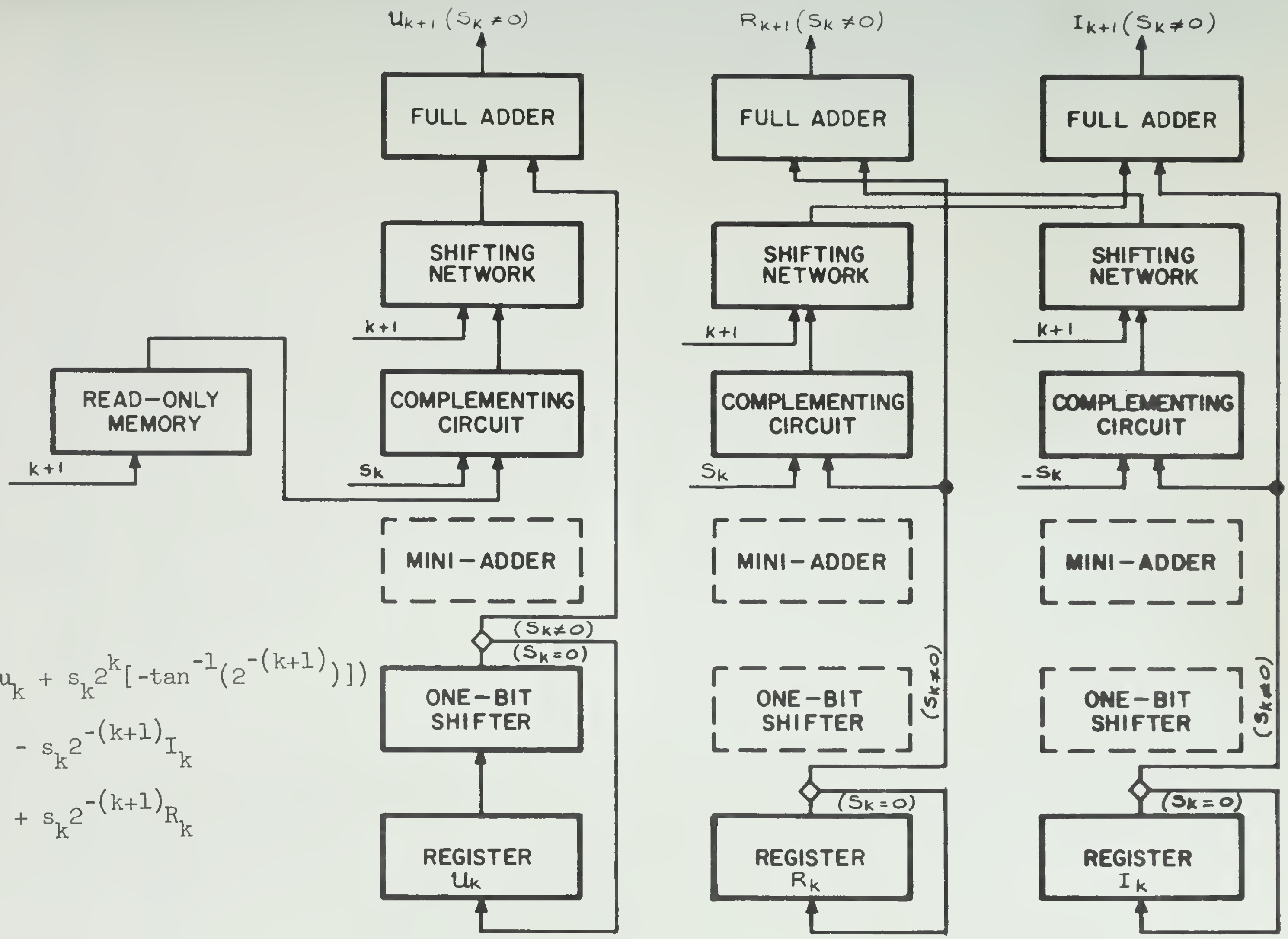
is made with an error of magnitude less than

$$\left|x_{M+1}\, e^{j\,x_{M+1}} \prod_{i=0}^{M} \left(\frac{t_i}{|t_i|}\right)\right|$$

or less than

$$|x_{M+1}|\,|e^{j\,x_{M+1}}|\,\left|\prod_{i=0}^{M} \left|\frac{t_i}{t_i}\right|\right|.$$

Hence, the errors in the values of cosine and sine are less than $\tan^{-1}(2^{-(M+1)})$ which is less than $2^{-(M+1)}$.



$$u_{k+1} = 2(u_k + s_k 2^k[-\tan^{-1}(2^{-(k+1)})])$$

$$R_{k+1} = R_k - s_k 2^{-(k+1)} I_k$$

$$I_{k+1} = I_k + s_k 2^{-(k+1)} R_k$$

FIGURE 10. Block diagram for cosine/sine

10.4 Experimental estimate of speed

Counting a non-zero step in the second quarter of the algorithm as two addition cycle times, the Monte Carlo estimate of the mean probability of a zero is 0.465, with a corresponding shift average of 1.88.

10.5 Implementation

Except that the control is complicated somewhat by changing operation modes in the midst of the algorithm, and that during the second quarter of the algorithm a "multiplication" of R_k and I_k by the factor $(1 - 2^{-(2k+3)})$ may be required at the completion of a non-zero step, the implementation discussed in Section 9.5 suffices.

10.6 Concluding remark

It may be preferable to perform the entire first half of the algorithm in the non-redundant mode. If this were done, it would require approximately 2/3 M as compared to 7/12 M addition cycle times, beyond initial range reduction, to compute both cosine and sine.

REFERENCES

1 W. H. Specker, "A Class of Algorithms for Ln X, Exp X, Sin X, Cos X, Tan^{-1} X, and Cot^{-1} X," IEEE Transactions on Electronic Computers, EC-14:1:85-86, February, 1965.

2 J. E. Volder, "The CORDIC Trigonometric Computing Technique," IEEE Transactions on Electronic Computers, EC-8:5:330-334, September, 1959.

11. THE ALGORITHM FOR ARCTANGENT

11.1 Basic algorithm

Formulating a normalization scheme required to force an arbitrarily large operand to zero in a few simple steps seems a rather formidable task, but, in this case, it is not difficult. A common trigonometric identity is useful.

$$\tan^{-1} X = \pi/4 + \tan^{-1}\left(\frac{X-1}{X+1}\right)$$

where

$$X = x \cdot 2^{\alpha} \qquad x \in [1/2,\ 1)$$

$$\alpha \text{ integer.}$$

Let

$$S = \begin{cases} 0 & \text{if} \quad \alpha \leq 0 \\ 1 & \text{if} \quad \alpha > 0 \end{cases}$$

and consider the following identity.

$$2^{-S\alpha}\left[(X+1) + j(X-1)\right] = \frac{\left[2^{-S\alpha}(X+1) + j\, 2^{-S\alpha}(X-1)\right] \prod_{i=0}^{M} t_i}{\prod_{i=0}^{M} t_i}$$

where the set of multipliers $\{t_i\}$ is of the same <u>form</u> as that used in the cosine/sine and tangent algorithms. Taking logarithms on both sides yields,

$$\ell n(2^{-S\alpha}) + \ell n\left(\sqrt{(X+1)^2 + (X-1)^2}\right) + j \tan^{-1}\left(\frac{X-1}{X+1}\right)$$

$$= \log(T_{M+1}) - \sum_{i=0}^{M} \ell n|t_i| - j \sum_{i=0}^{M} \varphi_i$$

where

$$T_{M+1} = R_{M+1} + j\, I_{M+1} = \left[2^{-S\alpha}(X+1) + j\, 2^{-S\alpha}(X-1)\right] \prod_{i=0}^{M} t_i$$

and

$$\log(T_{k+1}) = \ell n\left(\sqrt{R_{M+1}^2 + I_{M+1}^2}\right) + j \tan^{-1}\left(\frac{I_{M+1}}{R_{M+1}}\right).$$

Equating imaginary parts, one obtains,

$$\tan^{-1}\left(\frac{X-1}{X+1}\right) = \tan^{-1}\left(\frac{I_{M+1}}{R_{M+1}}\right) - \sum_{i=0}^{M} \varphi_i .$$

Hence, the desired relation is obtained.

$$\tan^{-1} X = \pi/4 + \tan^{-1}\left(\frac{I_{M+1}}{R_{M+1}}\right) - \sum_{i=0}^{M} \varphi_i .$$

Although $t_k = 1 + j\, 1/2\, s_k 2^{-k}$ as before, the selection rules for s_k are now chosen such that I_{M+1}/R_{M+1} is very nearly zero, so that one may approximate,

$$\tan^{-1} X \cong \pi/4 - \sum_{i=0}^{M} \varphi_i .$$

Three things should be noted carefully: (1) the required set of stored constants, $\{\tan^{-1}(2^{-(i+1)})\}$, is already required by other algorithms in the set; (2) although a division has been indicated <u>conceptually</u> in the transformation outlined above, no actual division need be performed--one merely sets

$$R_{00} = 2^{-S\alpha}(X+1)$$

$$I_{00} = 2^{-S\alpha}(X-1),$$

which requires less than one addition cycle time, (this is only the first part

of the initialization); (3) the algorithm, when implemented in hardware, is virtually the same as the algorithm for tangent, the only significant difference lying in that the quantity being normalized (indirectly) is the ratio $\left(\frac{X-1}{X+1}\right)$, rather than X itself. Comparisons to choose each s_k are performed with respect to I_k, and it is shown that $|I_{M+1}/R_{M+1}|$ is very small, so that

$$\left|\tan^{-1}\left(\frac{I_{M+1}}{R_{M+1}}\right)\right| \ll \frac{\pi}{4}$$

and the former term can be neglected. Never is the ratio I_{M+1}/R_{M+1} actually computed; rather, it is shown that $|I_{M+1}| \leq 3/8 \cdot 2^{-M}$, whereas $R_{M+1} \geq 3/4$.

As in the other trigonometric algorithms, three recursion relations are required.

$$A_{k+1} = A_k - \varphi_k \tag{11-1}$$

$$R_{k+1} = R_k - s_k I_k \, 2^{-(k+1)} \tag{11-2}$$

$$I_{k+1} = I_k + s_k R_k \, 2^{-(k+1)} \tag{11-3}$$

where

$$s_k = \begin{cases} 1 & \text{if} \quad I_k < -3/8 \cdot 2^{-k} \\ 0 & \text{otherwise} \\ \bar{1} & \text{if} \quad I_k \geq +3/8 \cdot 2^{-k}. \end{cases}$$

Understanding of the details of initialization and error bound proof is facilitated by discussing separately two cases: (1) $S = 0$, $0 \leq X < 1$; (2) $S = 1$, $1 \leq X < \infty$.

11.2 Choice of initialization--error bound: Case 1

The first part of the initialization consists of the choice of $S = 0$ and the setting of values

$$R_{00} = X + 1, \qquad R_{00} \in [1, 2)$$

$$I_{00} = X - 1, \qquad I_{00} \in [-1, 0).$$

The second part of the initialization, for this case, consists of a simple scaling of magnitudes.

$$t_{00} = \begin{cases} 3/4 & \text{if} \quad 0 \leq X < 1/4 \\ 5/8 & \text{if} \quad 1/4 \leq X < 1/2 \\ 1/2 & \text{if} \quad 1/2 \leq X < 1. \end{cases}$$

Note that $\varphi_{00} = 0$ and that $R_0 = R_{00}t_{00}$, $I_0 = I_{00}t_{00}$, can be formed in one addition cycle time. The third and last part of the initialization consists of beginning the recursion counter at $k = 0$, rather than $k = 1$ as in previous algorithms. (Considering this step as part of the initialization avoids the change in notation that would otherwise be implied.) By direct computation, it may be shown that $R_0 \in [3/4, 1)$, $I_0 \in [-3/4, 0) \in [-3/4, 3/4)$, which is sufficient to lead to a convergent algorithm.

<u>Hypothesis</u>: For $k \geq 0$,

$$-3/4 \cdot 2^{-k} \leq I_k \leq 3/4 \cdot 2^{-k}$$

$$3/4 + f(k) \leq R_k \leq 1 + 2f(k)$$

where

$$f(k) = 3/4 \sum_{i=0}^{k} |s_{i-1}| 2^{-2i}, \qquad s_{-1} = 0.$$

<u>Proof</u>: It was shown above that the hypothesis is true for $k = 0$. The induction proof is completed by considering the three ranges of I_k.

<u>Range 1</u>: Suppose $-3/4 \cdot 2^{-k} \leq I_k < -3/8 \cdot 2^{-k}$ so that $s_k = 1$ and

$$R_{k+1} = R_k - I_k 2^{-(k+1)}$$

$$I_{k+1} = I_k + R_k 2^{-(k+1)}$$

then,

$$3/4 + f(k) + 3/16 \cdot 2^{-2k} \leq R_{k+1} \leq 1 + 2f(k) + 3/8 \cdot 2^{-2k}$$

$$-3/4 \cdot 2^{-k} + (3/4 + f(k))\, 1/2 \cdot 2^{-k} \leq I_{k+1} < -3/8 \cdot 2^{-k} + (1 + 2f(k)) 1/2 \cdot 2^{-k}$$

thus,

$$3/4 + f(k+1) \leq R_{k+1} \leq 1 + 2f(k+1)$$

$$-3/8 \cdot 2^{-k} + 1/2 \cdot 2^{-k} f(k) \leq I_{k+1} \leq 1/8 \cdot 2^{-k} + 2^{-k} f(k).$$

Since $0 \leq f(k) < 1/4$,

$$3/4 + f(k+1) \leq R_{k+1} \leq 1 + 2f(k+1)$$

$$-3/8 \cdot 2^{-k} \leq I_{k+1} \leq +3/8 \cdot 2^{-k}$$

as desired.

Range 2: Suppose $-3/8 \cdot 2^{-k} \leq I_k < +3/8 \cdot 2^{-k}$ so that $s_k = 0$ and $R_{k+1} = R_k$, $I_{k+1} = I_k$. Since when $s_k = 0$, $f(k+1) = f(k)$, the hypothesis continues to hold in this range also.

Range 3: Suppose $3/8 \cdot 2^{-k} \leq I_k \leq 3/4 \cdot 2^{-k}$ so that $s_k = \bar{1}$ and

$$R_{k+1} = R_k + I_k 2^{-(k+1)}$$

$$I_{k+1} = I_k - R_k 2^{-(k+1)}$$

then,

$$3/4 + f(k) + 3/16 \cdot 2^{-2k} \leq R_{k+1} \leq 1 + 2f(k) + 3/8 \cdot 2^{-2k}$$

$$3/8 \cdot 2^{-k} - (1 + 2f(k))1/2 \cdot 2^{-k} \leq I_{k+1} \leq 3/4 \cdot 2^{-k} - (3/4 + f(k)) \cdot 1/2 \cdot 2^{-k}$$

thus,

$$3/4 + f(k+1) \leq R_{k+1} \leq 1 + 2f(k+1)$$

$$-1/8 \cdot 2^{-k} - 2^{-k} f(k) \leq I_{k+1} \leq 3/8 \cdot 2^{-k} - 1/2 f(k)2^{-k}.$$

But $0 \leq f(k) < 1/4$, so

$$3/4 + f(k+1) \leq R_{k+1} \leq 1 + 2f(k+1)$$

$$-3/8 \cdot 2^{-k} \leq I_{k+1} \leq 3/8 \cdot 2^{-k}$$

as desired.

Therefore after M steps,

$$|I_{M+1}| \leq 3/8 \cdot 2^{-M}$$

$$R_{M+1} \geq 3/4 + f(M+1) \geq 3/4$$

so that

$$\left|\frac{I_{M+1}}{R_{M+1}}\right| \leq 2^{-(M+1)}$$

and

$$\left|\tan^{-1}\left(\frac{I_{M+1}}{R_{M+1}}\right)\right| < 2^{-(M+1)}.$$

The error in the algorithm is thus small enough, in this case, to guarantee M correct bits in the approximation to $\tan^{-1} X$.

<u>Example</u>: As seen in Table 16, the choice of an operand $X = 0.6$ leads to a very quickly convergent algorithm as an approximation accurate to 14 decimal places is produced in only two steps beyond the initialization.

TABLE 16

k	s_k	A_{k+1}	R_{k+1}	I_{k+1}	I_{k+1}/R_{k+1}
1	0	0.78539816339745	0.80000000000000	-0.20000000000000	-0.25000000000000
2	1	0.54041950027058	0.85000000000000	0.00000000000000	0.00000000000000
3	0	0.54041950027058	0.85000000000000	0.00000000000000	0.00000000000000
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
40	0	0.54041950027058	0.85000000000000	0.00000000000000	0.00000000000000

11.3 Choice of initialization--error bound: Case 2

The first part of the initialization consists of the choice of $S = 1$ and the setting of values

$$R_{00} = 2^{-\alpha}(X+1) = x + 2^{-\alpha}, \quad R_{00} \in (1/2, 3/2)$$

$$I_{00} = 2^{-\alpha}(X-1) = x - 2^{-\alpha}, \quad I_{00} \in [0, 1).$$

The second part of the initialization consists of choosing t_{01}:

$$t_{01} = \begin{cases} 1 & \text{if} \quad 1 \leq X \leq 3/2 \\ 1\text{-}j1 & \text{if} \quad X > 3/2. \end{cases}$$

The third part of the initialization consists of a scaling of magnitudes.

$$t_{02} = \begin{cases} 3/4 & \text{if} \quad 1 \leq R_{01} < 5/4 \\ 5/8 & \text{if} \quad 5/4 \leq R_{01} < 3/2 \\ 1/2 & \text{if} \quad 3/2 \leq R_{01} < 2 \end{cases}$$

where, from the previous operation,

$$R_{01} \in [1, 2)$$

$$I_{01} \in [-1/2, 1/4).$$

Thus, by direct computation,

$$R_0 \in [3/4, 1]$$

$$I_0 \in [-3/8, +3/16) \in [-3/4, + 3/4].$$

Therefore, by the argument of Case 1, the error is again bound by $2^{-(M+1)}$.

Example: If $X = 1.2$, then $\tan^{-1} X = 0.87605805059819$. The algorithm produces an approximation which is in error by less than 0.2×10^{-12}, within the error bound of 0.45×10^{-12}.

TABLE 16

k	s_k	A_{k+1}	R_{k+1}	I_{k+1}	I_{k+1}/R_{k+1}
1	0	0.78539816339745	0.82500000000000	0.07500000000000	0.09090909090909
2	0	0.78539816339745	0.82500000000000	0.07500000000000	0.09090909090909
3	0	0.78539816339745	0.82500000000000	0.07500000000000	0.09090909090909
4	-1	0.84781697339341	0.82968750000000	0.02343750000000	0.02824858757062
5	0	0.84781697339341	0.82968750000000	0.02343750000000	0.02824858757062
6	-1	0.86344070201388	0.83005371093750	0.01047363281250	0.01261801817701
7	-1	0.87125304307398	0.83013553619385	0.00398883819580	0.00480504450405
8	-1	0.87515927320595	0.83015111759305	0.00074612125754	0.00089877763425
9	0	0.87515927320595	0.83015111759305	0.00074612125754	0.00089877763425
10	-1	0.87613583539551	0.83015184622709	-0.00006457319323	-0.00007778479747
11	0	0.87613583539551	0.83015184622709	-0.00006457319323	-0.00007778479747
12	0	0.87613583539551	0.83015184622709	-0.00006457319323	-0.00007778479747
13	0	0.87613583539551	0.83015184622709	-0.00006457319323	-0.00007778479747
14	1	0.87607480023934	0.83015185016833	-0.00001390474559	-0.00001674964114
15	0	0.87607480023934	0.83015185016833	-0.00001390474559	-0.00001674964114
16	1	0.87605954145027	0.83015185038049	-0.00000123763361	-0.00000149085208
17	0	0.87605954145027	0.83015185038049	-0.00000123763361	-0.00000149085208
18	0	0.87605954145027	0.83015185038049	-0.00000123763361	-0.00000149085208
19	0	0.87605954145027	0.83015185038049	-0.00000123763361	-0.00000149085208
20	1	0.87605858777596	0.83015185038168	-0.00000044593916	-0.00000053717776

(Continued)

TABLE 16 (Continued)

k	s_k	A_{k+1}	R_{k+1}	I_{k+1}	I_{k+1}/R_{k+1}
21	1	0.87605811093880	0.83015185038189	-0.00000005009187	-0.00000006034061
22	0	0.87605811093880	0.83015185038189	-0.00000005009187	-0.00000006034061
23	0	0.87605811093880	0.83015185038189	-0.00000005009187	-0.00000006034061
24	1	0.87605805133415	0.83015185038189	-0.00000000061096	-0.00000000073596
25	0	0.87605805133415	0.83015185038189	-0.00000000061096	-0.00000000073596
26	0	0.87605805133415	0.83015185038189	-0.00000000061096	-0.00000000073596
27	0	0.87605805133415	0.83015185038189	-0.00000000061096	-0.00000000073596
28	0	0.87605805133415	0.83015185038189	-0.00000000061096	-0.00000000073596
29	0	0.87605805133415	0.83015185038189	-0.00000000061096	-0.00000000073596
30	0	0.87605805133415	0.83015185038189	-0.00000000061096	-0.00000000073596
31	1	0.87605805086849	0.83015185038189	-0.00000000022439	-0.00000000027030
32	1	0.87605805063566	0.83015185038189	-0.00000000003111	-0.00000000003747
33	0	0.87605805063566	0.83015185038189	-0.00000000003111	-0.00000000003747
34	0	0.87605805063566	0.83015185038189	-0.00000000003111	-0.00000000003747
35	1	0.87605805060656	0.83015185038189	-0.00000000000694	-0.00000000000837
36	0	0.87605805060656	0.83015185038189	-0.00000000000694	-0.00000000000837
37	1	0.87605805059928	0.83015185038189	-0.00000000000090	-0.00000000000109
38	0	0.87605805059928	0.83015185038189	-0.00000000000090	-0.00000000000109
39	0	0.87605805059928	0.83015185038189	-0.00000000000090	-0.00000000000109
40	1	0.87605805059837	0.83015185038189	-0.00000000000015	-0.00000000000018

11.4 Experimental estimate of speed

Because of the sequence of steps in the initialization, the efficiency of the algorithm is somewhat less than desired, although the time required to compute $\tan^{-1} X$, $0 \le X < \infty$, is still only slightly greater (about three addition cycle times), than the time required to perform a division. For the steps beyond initialization, the mean probability of a zero is 0.650, with a corresponding shift average of 2.87.

11.5 Implementation

As has been the usual practice, a simple transformation, $u_k = 2^k I_k$, is made in order that the comparison constant may remain $\pm 3/8$, rather than $\pm 3/8 \cdot 2^{-k}$. Introducing this transformation into the recursion relations obtained earlier yields the following.

$$A_{k+1} = A_k - s_k \tan^{-1}(2^{-(k+1)}) \tag{11-4}$$

$$R_{k+1} = R_k - s_k 2^{-(2k+1)} u_k \tag{11-5}$$

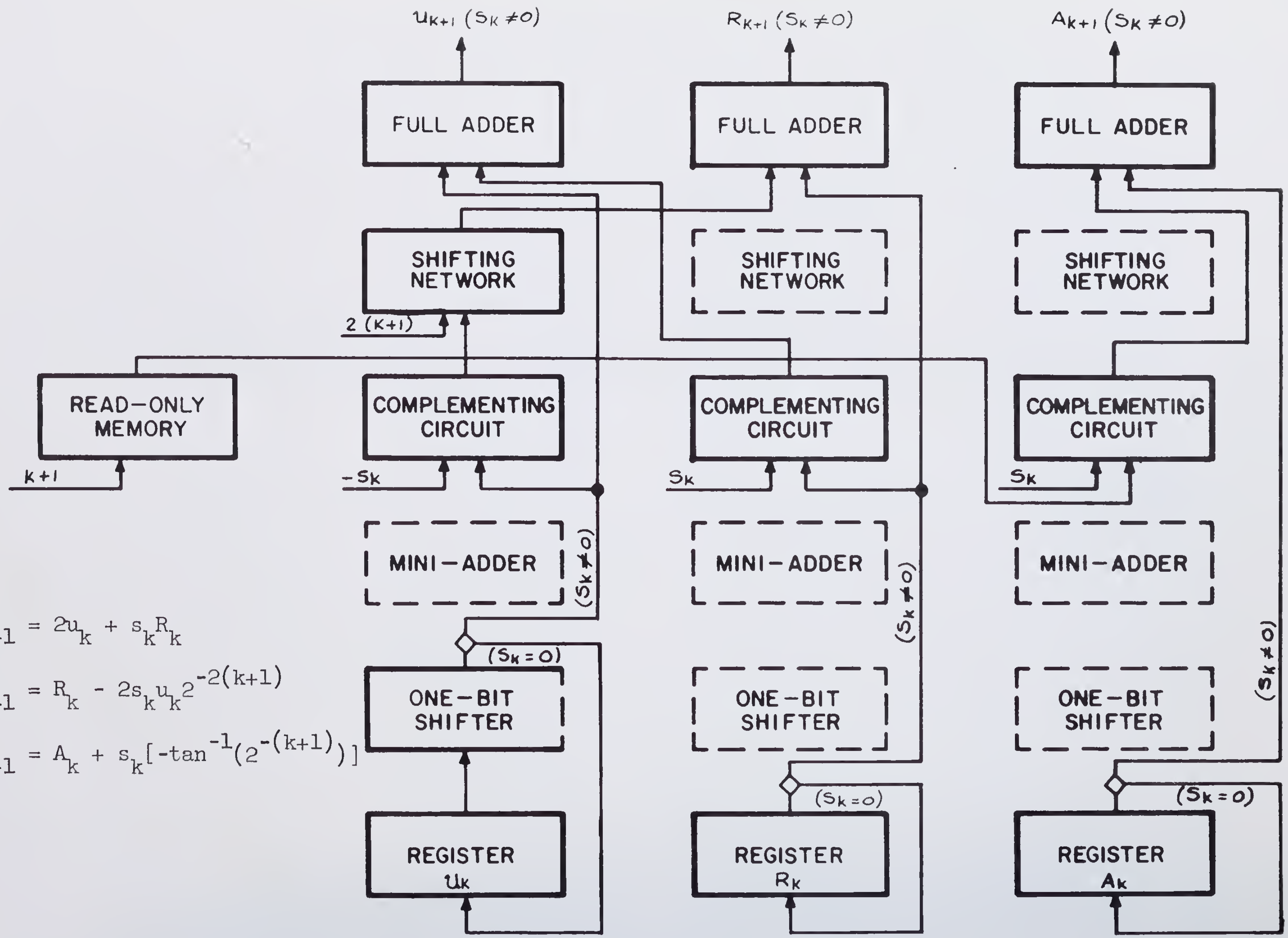
$$u_{k+1} = 2u_k + s_k R_k \tag{11-6}$$

It is of interest to note that, since $|u_k| < 1$, R_k remains fixed during the last half of the algorithm.

Figure 11 shows a block diagram to implement these recursions.

11.6 Concluding remark

Since the inverse cosine and inverse sine algorithms employ a version of the inverse tangent algorithm, it is important that this latter algorithm be as fast as possible--it is nearly as fast as division.



$$u_{k+1} = 2u_k + s_k R_k$$

$$R_{k+1} = R_k - 2s_k u_k 2^{-2(k+1)}$$

$$A_{k+1} = A_k + s_k[-\tan^{-1}(2^{-(k+1)})]$$

FIGURE 11. Block diagram for arctangent

12. A NOTE ON EVALUATION OF ARCCOSINE/ARCSINE

Previously developed algorithms suffice to evaluate arccosine or arcsine for $X \in [0, 1]$, although a modification of the initialization is required. From the identities

$$\sin^{-1} X = \tan^{-1}\left(\frac{X}{\sqrt{1-X^2}}\right) \tag{12-1}$$

$$\cos^{-1} X = \tan^{-1}\left(\frac{\sqrt{1-X^2}}{X}\right) \tag{12-2}$$

$$\sin^{-1} X + \cos^{-1} X = \pi/2 \tag{12-3}$$

it may be seen that a single multiplication to form X^2, followed by an application of a square root algorithm to form $\sqrt{1 - X^2}$, followed by the arctangent algorithm with a special initialization procedure to avoid the indicated division may be used to evaluate arccosine/arcsine. The initialization is simplified by considering two cases.

Case 1: If $\sqrt{1-X^2} \geq X$, $(0 \leq X \leq 1/\sqrt{2})$, then set

$$R_{00} = \sqrt{1-X^2} \quad \in [1/\sqrt{2}, 1]$$

$$I_{00} = X \quad \in [0, 1/\sqrt{2}]$$

and use (12-1) to compute $\sin^{-1} X$ and (12-3) to compute $\cos^{-1} X$ if desired. The next part of the initialization consists of choosing $t_{01} = 1 - j\,1/2$ so that

$$R_{01} = \sqrt{1-X^2} + 1/2\, X$$

$$I_{01} = X - 1/2 \sqrt{1-X^2}.$$

Finally choose $t_{02} = 3/4$ to scale magnitudes to desirable ranges.

$$R_0 = 3/4\ R_{01} \in [3/4,\ 1]$$

$$I_0 = 3/4\ I_{01} \in [-3/8,\ +3/8] \in [-3/4,\ +3/4].$$

Continuation of the arctangent algorithm then provides the desired result with an error less than $2^{-(M+1)}$.

<u>Case 2</u>: If $\sqrt{1-X^2} < X$, $(1/\sqrt{2} < X \leq 1)$, then set

$$R_{00} = X \in (1/\sqrt{2},\ 1]$$

$$I_{00} = \sqrt{1-X^2} \in [0, 1/\sqrt{2})$$

and continue as in Case 1.

It may thus be concluded that arccosine/arcsine may be evaluated in slightly more than three multiplication cycle times.

13. ON A HIGHER RADIX IMPLEMENTATION

13.1 General considerations

One of the chief limitations on the speed with which the algorithms developed in this research may be implemented is the step-by-step transposition in the control: s_k is chosen, then the appropriate recursions performed, s_{k+1} is chosen, the recursions performed, and so on. If the control time is at all significant, there are clear speed advantages in choosing not only s_k but also s_{k+1}, ... from comparisons on u_k and then performing several (binary) steps in the recursion relations. The cost of this higher radix implementation is a significant complication of the required comparisons to choose s_{k+1}, ... In general, both an increase in the number of comparisons required and an increase in the precision of the comparisons would be expected.

Although higher radix implementations are not a subject of detailed study in this research, a few apparently important considerations are known and it is desirable to discuss them here. Only radices which are integer powers of two are considered.

To illustrate the general strategy, let us compare the recursion for the division scheme of Section 3 necessary to choose s_k, s_{k+1}, ... with the analogous recursion for most other division schemes.

$$u_{k+1} = 2u_k + s_k + 2^{-k}s_k u_k \tag{13-1}$$

$$u_{k+1} = 2u_k - s_k x \tag{13-2}$$

The symbols in (13-1) have been previously defined; in (13-2), s_k is the k^{th} quotient digit, u_k and u_{k+1} are partial remainders (u_0 is the dividend), and x is the divisor. In either (13-1) or (13-2) the range of u_{k+1} is the same

as the range of u_k. Except for the first few steps, the term $2^{-k}s_k u_k$ in (13-1) contains information of only secondary importance, that is, the dominant digits of u_{k+1} are determined only by $2u_k$ and s_k (recall $s_k = \{\bar{1}, 0, 1\}$). Thus, from the value $2u_k$ alone, one is able to choose not only s_k, but also s_{k+1}, ... From (13-2) it is seen that, independent of the step index k, significant information is contained in the term $s_k x$, and schemes for higher radix implementation must take this effect into account, that is, the recoding is a function of the divisor. (Scaling procedures have been studied to overcome this difficulty in the division represented by (13-2)). To facilitate the implementation of a higher radix procedure, one would prefer to have a recursion which does not explicitly depend on the original operand--except for the starting difficulty incurred, recursion (13-1) is considerably less complicated to implement in radix $r = 2^n$, $n > 1$, than is recursion (13-2).

Most, but not all, of the algorithms developed in earlier sections of this paper lend themselves to such a simplified higher radix implementation.

13.2 Amenability of normalization algorithms to higher radix

Once the first few steps have been performed, the division scheme proposed in Section 3 lends itself well to a higher radix implementation. The multiplication algorithm of Section 4 requires the simple recursion

$$u_{k+1} = 2\,u_k - s_k \qquad (13\text{-}3)$$

and a higher radix implementation is visibly easier for multiplication than for division since no starting problem exists (beyond the actual initialization).

Two square root algorithms were discussed in detail in Sections 6 and 7; the recursions of present interest are listed below.

$$u_{k+1} = (2u_k + s_k) + 2^{-k}(2s_k u_k + 1/4\ s_k^2) + 2^{-2k}(1/2\ s_k^2\ u_k) \qquad (13\text{-}4)$$

$$u_{k+1} = 2u_k \div 1/2(s_k R_k + s_k^2\ 2^{-(k+2)}) \qquad (13\text{-}5)$$

In the strictly binary case, the second algorithm, represented by recursion (13-5), is clearly superior since it is less complex and can be implemented with less time and/or hardware. However, it has the disadvantage of being more difficult to implement in higher radix because the recursion is a strong function of R_k, the approximation to $\sqrt{x}$. Thus, this algorithm presents at least the same level of difficulty in a higher radix implementation as the division represented by (13-2). The first square root algorithm, represented by (13-4), is clearly comparable to the division in (13-1).

It is perhaps less obvious but still easy to show that the algorithm for exponential is readily amenable to higher radix.

$$\begin{aligned} u_{k+1} &= 2u_k - 2^{k+1}\ell n(1 + 2^{-(k+1)}s_k) \\ &= 2u_k - 2^{k+1}[2^{-(k+1)}s_k - 1/2\ 2^{-2(k+1)}s_k^2 \\ &\qquad + \text{higher order terms}] \\ &= (2u_k - s_k) + s_k^2\ 2^{-(k+2)} + \text{higher order terms} \qquad (13\text{-}6) \end{aligned}$$

Again, after some difficulty getting started, a higher radix implementation may be feasible. The same is true of the algorithms for tangent and cosine/sine.

$$\begin{aligned} u_{k+1} &= 2u_k - s_k 2^{k+1}\tan^{-1}(2^{-(k+1)}) \\ &= 2u_k - s_k 2^{k+1}[2^{-(k+1)} - 1/3\ 2^{-3(k+1)} + \text{higher order terms}] \\ &= (2u_k - s_k) + 1/3\ s_k 2^{-2(k+1)} + \text{higher order terms.} \qquad (13\text{-}7) \end{aligned}$$

For the arctangent algorithm, the issue is less clear. From the recursion,

$$u_{k+1} = 2u_k + s_k R_k \tag{13-8}$$

it appears that this algorithm is even more difficult to implement in higher radix than the division of (13-2) since R_k is not only a function of the given operand, but a function of the step index as well. However, it was shown in Section 11 that

$$R_k = K + g(k)$$

where $K \in [3/4, 1)$ is a function only of the original operand and $g(k)$ is a very slowly changing function of the step index and is a second-order effect beyond the first few steps. Hence, (13-8), rewritten as,

$$u_{k+1} = 2u_k + s_k K + \text{higher order terms} \tag{13-9}$$

appears to present the same level of difficulty in higher radix as does (13-2).

From these general preliminary considerations, all of the algorithms except the second square root and the arctangent appear to be readily amenable to higher radix implementation.

13.3 A change in strategy

One may view a higher radix implementation, say radix 4, as a simple alteration of control in that two successive values s_k, s_{k+1} are chosen on the basis of u_k alone without forming u_{k+1}. Even though the probability that $s_k = 0$ may approach 2/3, the probability of a radix 4 digit, represented by $s_k s_{k+1}$, being zero is quite small. Furthermore, the digit

sequences 11 and $\bar{1}\bar{1}$, representing values 3 and $\bar{3}$, are to be avoided since two additions (or subtractions) are required. In a radix 4 implementation, one may wish to limit redundancy by allowing only digital values $\bar{2}$, $\bar{1}$, 0, 1, 2.

Since the probability of a zero in radix 4 is small (about 1/4 if values $\bar{2}$, $\bar{1}$, 0, 1, 2 are allowed) and speed is achieved by reducing control time, the shift average is no longer a meaningful measure of efficiency. Rather the speed itself or a speed to hardware ratio must be extablished.

While no efficiency studies of radix 4 implementations of these algorithms have been made, a few thoughts in that direction are appropriate.

13.4 Efficiency of radix 4 versus radix 2

For as specific a comparison as can be made at this time, let us consider the multiplication algorithm of Section 4 which presents no starting problems for a higher radix implementation. The time required to perform multiplication radix 2 is given by

$$T_2 \cong M[t_C + 1/3\ t_A + t_S + 1/3\ t_{SH}]$$

where

t_C = control sequencing time

t_A = addition/subtraction time (including complementation)

t_S = selection time--time required to select a value for s_k

t_{SH} = shifting time.

The time required to perform multiplication radix 4 is given by

$$T_4 \cong M/2[\alpha\ t_C + 3/4\ t_A + \beta\ t_S + 4/5\ t_{SH}]$$

where α is a measure of the complication of the control and β is a measure of the complication in the selection rules caused by choosing two "multipliers"

at once; probably $\alpha \sim 1$, $\beta \sim 2$. It has been assumed throughout this work that complementations, low precision comparisons, and shifting operations are much faster than addition. Thus,

$$T_2 \cong M\,[t_C + 1/3\ t_A]$$

$$T_4 \cong M\,[1/2\ t_C + 3/8\ t_A].$$

It thus appears that if the control time is at all appreciable relative to the addition time, a radix 4 implementation would be faster than radix 2: $T_2 > T_4$ if $t_c > 1/12\ t_A$.

Similar considerations, but not necessarily similar conclusions, appear to apply to any radix $r = 2^n$, $n \geq 2$.

14. CONCLUSIONS

14.1 A set of algorithms

A class of algorithms for evaluation of certain elementary functions, suitable for hard-wire implementation in a scientific binary digital computer, has been developed and studied. It has been shown that all of the algorithms can be implemented with a reasonably economical structure, and it is strongly believed that these algorithms would be considerably faster than software routines that are now often used. The algorithms employ minimal redundancy to achieve an increase in speed over non-redundant versions of the same algorithms; the cost of employing redundancy is a complication of the initialization of the algorithms and an increase in the precision of the comparisons required to choose multiplier constants.

The extension of the algorithms to higher radix, restricted to the case where the radix is a power of two, has been briefly studied with the observation that the recursion upon which s_k, s_{k+1}, ... are based should be independent of the initial operand. With the exception of the arctangent, algorithms are known for every function in the set that have this favorable property. In contrast, the recursion commonly used for division,

$$u_{k+1} = r\,u_k - q_k\,x$$

where q_k is the k^{th} radix r quotient digit, u_k and u_{k+1} are partial remainders (u_0 is the dividend), and x is the divisor, does not have this favorable property.

The application of redundant arithmetic recodings has been extended to a limited set of functions. It is possible that much broader classes of functions may lend themselves to approaches similar to those employed here.

14.2 Areas of further investigation

14.2.1 Generalization of the technique

The generalization and extension of the "normalization" technique presents, probably, the most interesting topic for further investigation. It is known that either a continued product or a continued summation representation of a divisor or its reciprocal yields two possible division algorithms. Similarly it is known that either representation leads to a pair of algorithms for square root. Surprisingly, only a single such algorithm for multiplication appears to offer any practicality. Only a single algorithm is known for each of the other functions studied.

The success in devising the algorithms discussed in this report is due basically to three useful properties:

(1) $\log (\Pi\ p_i) = \Sigma \log p_i$

(2) $\exp (\Sigma\ s_i) = \Pi(\exp s_i)$

(3) $\exp (\log x) = \log (\exp x) = x.$

It is not known what general class of functions may be evaluated through such a formulation, or whether that class has already been exhausted in this paper. The hyperbolic functions seem likely candidates, but were not studied here because they can quite easily be formed in terms of exponentials.

14.2.2 Correspondence between normalization recodings and classical multiplier recodings

Considerable attention has been devoted to the study of multiplier recodings and the correspondence between certain digital division techniques and these recodings.[1,2] It is clear that a similar, but perhaps less direct, correspondence exists between the recodings of these algorithms and the multiplier recodings. A complete analysis of this correspondence would be quite valuable.

14.2.3 <u>Some partial insight?</u>

Perhaps some insight into both of the problems discussed above is available even now.

Given a number in conventional form,

$$x = \sum_{i=0}^{\infty} s_i 2^{-i}, \qquad s_i \in \{\bar{1}, 0, 1\} \tag{14-1}$$

one may recode as

$$x = \sum_{i=0}^{\infty} s_i' \, 2^{-i}, \qquad s_i' \in \{\bar{1}, 0, 1\} \tag{14-2}$$

as in the division scheme. However, more generally, one may recode as

$$x = \sum_{i=0}^{\infty} s_i'' w(i), \qquad s_i'' \in \{\bar{1}, 0, 1\} \tag{14-3}$$

where $w(i)$ represents a "weighting function" not necessarily equal to its "nominal" value of 2^{-i}. Note that this is no longer a strictly radix 2 recoding, although it is assumed that $w(i)$ is on the order of 2^{-i}. In particular, it is known that

$$s_i w(i) = s_i \tan^{-1}(2^{-i}) \tag{14-4}$$

and

$$s_i w(i) = \ell n(1 + s_i 2^{-i}) \tag{14-5}$$

have practical application. In attempting to determine what class of functions may be evaluated by a normalization algorithm, or in studying the properties of the resulting recodings, it is natural to seek properties of weight functions that produce algebraically correct recodings. (It is not to be inferred that merely because a set of weights produces an algebraically correct recoding the set has any practical application, but it is assumed to be a necessary prerequisite.)

It is convenient, first of all, that the weights be positive and be ordered:

$$0 < w(i+1) < w(i). \tag{14-6}$$

It would appear at first glance that another convenient property would be that the ratio of successive weights

$$\frac{w(i+1)}{w(i)}$$

should be constant, but some of those weights found to have practical application do not satisfy this property. If digital values are limited to the set $\{\bar{1}, 0, 1\}$, then one would certainly like to have the property that the k^{th} weight satisfies

$$w(k) \leq \sum_{i=k+1}^{\infty} w(i). \tag{14-7}$$

The special class of weights

$$w(i) \leq \beta^{-i}, \quad 1 < \beta \leq 3 \tag{14-8}$$

may be seen to be acceptable, but this class does not include those sets of weights found to have practical application. A second-order perturbation solves this problem:

$$w(i) = \beta^{-i} - f(i), \quad 1 < \beta \leq 3 \tag{14-9}$$

where

$$|f(i)| \ll \beta^{-i}$$

$$w(i)^{*} \leq 3^{-i}.$$

---

* With only three digital values, one cannot recode in a radix greater than three.

With this perturbation, the weights represented in equations (14-4) and (14-5) fall into this class. The classes of weights represented by (14-8) and (14-9) also suggest the property that

$$\frac{w(i+1)}{w(i)} \in [1/3, 1).$$

Certainly no answers to the problems discussed above have been given, but perhaps an insight into a direction of study has been given.

14.2.4 A different radix 4 approach

In Section 13 it was suggested that one method of generating a radix 4 implementation was to simply choose two successive binary digital values s_k, s_{k+1} at once, probably limiting the radix 4 digital value $s_{k,\ k+1}$ to one of the set $\{\bar{2}, \bar{1}, 0, 1, 2\}$. A natural additional step would be to change the weight functions to be of the form 4^{-i} or $\tan^{-1}(4^{-i})$ or $\ell n(1 + s_{i,\ i+1}4^{-i})$. Such a modification would eliminate roughly half of the stored constants.

14.2.5 Some practical matters

As practical matters, development of actual hardware control circuitry (or perhaps equivalent microprogram control) and an "optimization" of initialization steps remain for anyone seriously interested in implementing these algorithms in a machine.

REFERENCES

1 J. O. Penhollow, "A Study of Arithmetic Recoding with Applications in Multiplication and Division," University of Illinois DCL Report No. 128, September 10, 1962.

2 J. E. Robertson, "The Correspondence Between Methods of Digital Division and Multiplier Recoding Procedures," University of Illinois DCL Report No. 252, December 21, 1967.

APPENDIX A: MONTE CARLO ESTIMATE OF THE PROBABILITY OF A ZERO

A.1 Confidence in estimate of probability of a zero

Although the theoretical asymptotic value of the probability of a zero is known, one would, as a practical matter, have to have an estimate for a register of finite length; in particular, a register length of 40 was chosen.

Following the discussion given by Cochran,[1] let p_0 be the estimat probability of a zero and P_0 be the actual probability. Further, let d be the margin of error in p_0, and let α be the risk that the actual error is larger than d.

$$\alpha = \mathrm{Prob}(|p_0 - P_0| \geq d)$$

If p_0 is taken as normally distributed, then the standard deviation σ_{p_0} is given by

$$\sigma_{p_0} = \sqrt{\frac{N-n}{N-1}} \sqrt{\frac{P_0(1-P_0)}{n}}$$

where N is the population size ($N \cong 2^{40}$ for a register length of 40) and n the sample size ($n = 2^{18}$). The formula that connects n with the degree of precision is

$$d = t \sqrt{\frac{N-n}{N-1}} \sqrt{\frac{P_0(1-P_0)}{n}}$$

where t is the abscissa of the normal curve that cuts off an area α at the tails; solving for n,

$$n = \frac{\dfrac{t^2 P_0(1-P_0)}{d^2}}{1 + \dfrac{1}{N}\left(\dfrac{t^2 P_0(1-P_0)}{d^2} - 1\right)} .$$

For practical use, the estimate p_0 must be used. Since $n << N$,

$$n \cong \frac{t^2 p_0(1-p_0)}{d^2} = \frac{p_0(1-p_0)}{V}$$

where $V = d^2/t^2$ = variance of the sample. Thus, the sample variance is

$$V \cong \frac{p_0(1-p_0)}{n} \cong \frac{(2/3)(1/3)}{2^{18}} \cong 0.848 \times 10^{-6}$$

and the sample standard deviation is

$$\sqrt{V} \cong 0.920 \times 10^{-3}.$$

With 99% confidence, one may say that the actual deviation is within about $2.58\sqrt{V} \cong 2.37 \times 10^{-3}$; with 99.9% confidence, one may say that the actual deviation is within about $3.29\sqrt{V} \cong 3.03 \times 10^{-3}$. Thus, for a sample of 2^{18}, P_0 is known accurately enough for the purposes of this research.

A.2 Generation of pseudo-random double precision operands

The IBM 360/75 computer system software package at the University of Illinois contains a single precision pseudo-random number generator employing a version of Lehmer's method. The subroutine generates pseudo-random numbers using the recurrence relations

$$\alpha_{j+1} = 2^{-42}\beta_j$$

$$\beta_{j+1} = 5^{17}\beta_j (\text{mod } 2^{42}),\ \beta_0 = 1$$

with a period of approximately 2^{40}. Since a sample size of 2^{18} was chosen, each of the random operands in the sample is unique. Two single precision operands are required to provide each double precision operand.

As described by Schreider,[2] the chi-square and Kolmogorov tests of uniformity of random numbers are the most commonly accepted criteria for accepting or rejecting a set of pseudo-random numbers. A sample of ten sets of 2^{10} double precision pseudo-random numbers passed both of these tests.

A.2.1 The chi-square test

The chi-square test is based on the statistic

$$\chi^2 = \sum_{i=1}^{N} \frac{(\nu_i - np_i)^2}{np_i} ,$$

where ν_i is the sample number of objects in the i^{th} interval, np_i is the expected value of ν_i, and N is the number of intervals chosen. The values $N = 10$, $n = 2^{10}$, $p_i = 1/10$, $i = 1, 2, \ldots, 10$ were used. Thus,

$$\chi^2 = \sum_{i=1}^{10} \frac{(\nu_i - 102.4)^2}{102.4} .$$

The hypothesis of uniform distribution of the pseudo-random numbers is rejected if χ^2 exceeds the upper limit $\chi^2_{N-1(p)}$ of the confidence interval, where p is the assigned confidence probability* and N - 1 is the number of degrees of freedom. In this test, $p = 0.95$ was chosen, so that values of χ^2 should not exceed $\chi^2_{9(.95)} \cong 16.9$.

Extremely small values of χ^2 are considered an indication of failure of randomness. The critical region of acceptance of the hypothesis is taken to be

$$[\chi^2_{N-1(1-p)}, \chi^2_{N-1(p)}] \cong [3.33, 16.9].$$

---

* A confidence level p means a probability close to one such that, if the hypothesis of uniform distribution is correct, the probability is p that the value obtained for χ^2 will not exceed $\chi^2(p)$.

In the case of $p = 0.95$, the probability that χ^2 lies outside the above interval is far from negligible, about one trial in ten. Ten samples were tested; the values of χ^2 for the test samples are listed below. Since only one trial falls outside the critical region, the hypothesis is accepted.

TABLE 17

Trial	χ^2
1	6.90
2	7.80
3	5.36
4	11.41
5	4.18
6	4.22
7	13.21
8	13.62
9	21.43
10	9.09

A.2.2 The Kolmogorov Criterion

Kolmogorov's criterion is based on the statistic

$$D_n = \max\left|\frac{m_x}{n} - F(x)\right|$$

where $n = 2^{10}$ is the size of the sample, m_x is the number of objects in the sample not exceeding the value x, and F(x) is the theoretical cumulative probability function.

Schreider suggests rejecting the hypothesis of "uniformity" if $n^{1/2}D_n$ falls outside the range (0.5, 1.5). The table below lists the values of $n^{1/2}D_n$ for ten test trials.

TABLE 18

Trial	$n^{1/2}D_n$
1	0.537
2	0.744
3	0.319
4	0.613
5	0.638
6	0.519
7	0.713
8	0.825
9	1.119
10	0.956

Since for nine of ten trials the value of $n^{1/2}D_n$ falls in the desired range, the hypothesis is accepted. It may be noted qualitatively, however, that the data indicate that the numbers generated tend to be somewhat more nearly uniform than would be expected of purely random numbers.

A.2.3 Listing of raw data for statistical tests

TABLE 19

x	m_x*									
	1	2	3	4	5	6	7	8	9	10
0.1	109	105	100	92	108	88	101	128	100	97
0.2	218	199	195	191	211	191	200	227	185	203
0.3	319	301	299	297	313	291	308	330	307	307
0.4	422	401	418	418	430	393	401	436	422	419
0.5	529	504	516	520	529	499	506	531	508	538
0.6	618	605	617	634	620	607	611	618	593	645
0.7	709	693	708	721	722	711	715	715	681	735
0.8	802	815	809	812	819	821	842	812	802	836
0.9	918	927	915	927	924	915	920	931	908	940
1.0	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024

REFERENCES

1 W. G. Cochran, Sampling Techniques, pp. 49-86.

2 Y. A. Schreider, Method of Statistical Testing Monte Carlo Method, pp. 7-18, 196-220.

* Ten trials for m_x are listed.

APPENDIX B: DISCUSSION OF THE CHOICE OF MULTIPLIER CONSTANTS IN THE COSINE/SINE ALGORITHM

The cosine/sine algorithm employs the identity

$$e^{jx} = e^{j(x - \sum_{i=0}^{M} \varphi_i)} \frac{\prod_{i=0}^{M} t_i}{\prod_{i=0}^{M} |t_i|}$$

where $\{\varphi_i\}$ must be chosen such that

$$x - \sum_{i=0}^{M} \varphi_i \cong 0.$$

Since the cosine and sine are required, one must evaluate $T_k = R_k + j\, I_k$ where

$$R_k = \mathrm{Re} \left\{ \frac{\prod_{i=0}^{k-1} t_i}{\prod_{i=0}^{k-1} |t_i|} \right\}$$

$$I_k = \mathrm{Im} \left\{ \frac{\prod_{i=0}^{k-1} t_i}{\prod_{i=0}^{k-1} |t_i|} \right\} .$$

Given T_k, one must compute the real and imaginary parts of T_{k+1}. Let us consider the following.

(1) Write

$$T_{k+1} = \prod_{i=0}^{k} \frac{t_i}{|t_i|} = \prod_{i=0}^{k} (\cos \varphi_i + j \sin \varphi_i).$$

Then,

$$T_{k+1} = (R_k \cos \varphi_k - I_k \sin \varphi_k) + j(I_k \cos \varphi_k + R_k \sin \varphi_k).$$

Since it is not possible, in general, that both $\cos \varphi_k$ and $\sin \varphi_k$ have only one non-zero bit, this is not an efficient recursion to perform.

(2) Write

$$T_{k+1} = \prod_{i=0}^{k} (t_i \lambda_i)$$

$$\text{where } \lambda_i = \frac{1}{|t_i|}.$$

Then,

$$T_{k+1} = (R_k r_k - I_k i_k)\lambda_k + j(I_k r_k + R_k i_k)\lambda_k$$

where $t_k = r_k + j\, i_k$. Since $\lambda_k \sqrt{r_k^2 + i_k^2} = 1$, it is not possible, in general, that r_k, i_k, and λ_k each have only one non-zero bit. Hence, the recursion cannot be performed efficiently in this formulation.

(3) As a variant of the above, let us choose

$$r_k = 1$$

$$i_k = s_k\, 2^{-(k+1)}$$

so that

$$\lambda_k = \frac{1}{\sqrt{1 + s_k^2\, 2^{-2(k+1)}}}.$$

Then the necessary recursions can be performed if $s_k^2 = 1$ and $\Pi\, \lambda_i$ is <u>precomputed</u>. That for k sufficiently large, a simple approximation for λ_k suffices has been shown in Section 10.

APPENDIX C: LISTING OF PRECOMPUTED CONSTANTS

All precomputed constants required by the algorithms presented in this paper are listed below; sufficient accuracy for a mantissa of length 40 is retained.

TABLE 20

Special Constant	Approximate Value*
π	3.14159265358979
$\pi/2$	1.57079632679490
$\pi/4$	0.78539816339745
$\pi/8$	0.39269908169872
$1/\sqrt{2}$	0.70710678118655
K'	0.88706417837978
K''	0.36743401338025
tan $\pi/8$	0.41421356237310

* Note that the values of K' and K'' are functions of M.

TABLE 21

i	2^{-i}	3.2^{-i}
1	0.50000000000000	1.50000000000000
2	0.25000000000000	0.75000000000000
3	0.12500000000000	0.37500000000000
4	0.06250000000000	0.18750000000000
5	0.03125000000000	0.09375000000000
6	0.01562500000000	0.04687500000000
7	0.00781250000000	0.02343750000000
8	0.00390625000000	0.01171875000000
9	0.00195312500000	0.00585937500000
10	0.00097656250000	0.00292968750000
11	0.00048828125000	0.00146484375000
12	0.00024414062500	0.00073242187500
13	0.00012207031250	0.00036621093750
14	0.00006103515625	0.00018310546875
15	0.00003051757813	0.00009155273438
16	0.00001525878906	0.00004577636719
17	0.00000762939453	0.00002288818359
18	0.00000381469726	0.00001144409180
19	0.00000190734863	0.00000572204590
20	0.00000095367432	0.00000286102295
21	0.00000047683716	0.00000143051147
22	0.00000023841858	0.00000071525574
23	0.00000011920929	0.00000035762787
24	0.00000005960464	0.00000017881393
25	0.00000002980232	0.00000008940697

(Continued)

TABLE 21 (Continued)

i	2^{-i}	3.2^{-i}
26	0.00000001490116	0.00000004470348
27	0.00000000745058	0.00000002235174
28	0.00000000372529	0.00000001117587
29	0.00000000186265	0.00000000558794
30	0.00000000093132	0.00000000279397
31	0.00000000046566	0.00000000139698
32	0.00000000023283	0.00000000069849
33	0.00000000011642	0.00000000034925
34	0.00000000005821	0.00000000017462
35	0.00000000002910	0.00000000008731
36	0.00000000001455	0.00000000004366
37	0.00000000000728	0.00000000002183
38	0.00000000000364	0.00000000001091
39	0.00000000000182	0.00000000000546
40	0.00000000000091	0.00000000000273
41	0.00000000000045	0.00000000000136
42	0.00000000000023	0.00000000000068
43	0.00000000000011	0.00000000000034
44	0.00000000000006	0.00000000000017
45	0.00000000000003	0.00000000000009

TABLE 22

i	$\ln(1+2^{-i})$	$\ln(1-2^{-i})$
1	0.40546510810816	-0.69314718055995
2	0.22314355131421	-0.28768207245178
3	0.11778303565638	-0.13353139262452
4	0.06062462181643	-0.06453852113757
5	0.03077165866675	-0.03174869831458
6	0.01550418653597	-0.01574835696814
7	0.00778214044205	-0.00784317746103
8	0.00389864041566	-0.00391389932114
9	0.00195122013126	-0.00195503483580
10	0.00097608597306	-0.00097703964783
11	0.00048816207950	-0.00048840049811
12	0.00024411082753	-0.00024417043217
13	0.00012206286253	-0.00012207776369
14	0.00006103329368	-0.00006103701897
15	0.00003051711247	-0.00003051804380
16	0.00001525867265	-0.00001525890548
17	0.00000762936543	-0.00000762942364
18	0.00000381468999	-0.00000381470454
19	0.00000190734681	-0.00000190735045
20	0.00000095367386	-0.00000095367477
21	0.00000047683704	-0.00000047683727
22	0.00000023841855	-0.00000023841861
23	0.00000011920928	-0.00000011920930
24	0.00000005960464	-0.00000005960465
25	0.00000002980232	-0.00000002980232

(Continued)

TABLE 22 (Continued)

i	$\ln(1+2^{-i})$	$\ln(1-2^{-i})$
26	0.00000001490116	-0.00000001490116
27	0.00000000745058	-0.00000000745058
28	0.00000000372529	-0.00000000372529
29	0.00000000186265	-0.00000000186265
30	0.00000000093132	-0.00000000093132
31	0.00000000046566	-0.00000000046566
32	0.00000000023283	-0.00000000023283
33	0.00000000011642	-0.00000000011642
34	0.00000000005821	-0.00000000005821
35	0.00000000002910	-0.00000000002910
36	0.00000000001455	-0.00000000001455
37	0.00000000000728	-0.00000000000728
38	0.00000000000364	-0.00000000000364
39	0.00000000000182	-0.00000000000182
40	0.00000000000091	-0.00000000000091
41	0.00000000000045	-0.00000000000045
42	0.00000000000023	-0.00000000000023
43	0.00000000000011	-0.00000000000011
44	0.00000000000006	-0.00000000000006
45	0.00000000000003	-0.00000000000003

TABLE 23

i	arctan (2^{-i})
1	0.46364760900081
2	0.24497866312686
3	0.12435499454676
4	0.06241880999596
5	0.03123983343027
6	0.01562372862048
7	0.00781234106010
8	0.00390623013197
9	0.00195312251648
10	0.00097656218956
11	0.00048828121119
12	0.00024414062015
13	0.00012207031189
14	0.00006103515617
15	0.00003051757812
16	0.00001525878906
17	0.00000762939453
18	0.00000381469726
19	0.00000190734863
20	0.00000095367432
21	0.00000047683716
22	0.00000023841858
23	0.00000011920929
24	0.00000005960464
25	0.00000002980232

(Continued)

TABLE 23 (Continued)

i	arctan (2^{-i})
26	0.00000001490116
27	0.00000000745058
28	0.00000000372529
29	0.00000000186265
30	0.00000000093132
31	0.00000000046566
32	0.00000000023283
33	0.00000000011642
34	0.00000000005821
35	0.00000000002910
36	0.00000000001455
37	0.00000000000728
38	0.00000000000364
39	0.00000000000182
40	0.00000000000091
41	0.00000000000045
42	0.00000000000023
43	0.00000000000011
44	0.00000000000006
45	0.00000000000003

APPENDIX D: LISTING OF PARTIAL SIMULATION PROGRAM CODE

```
// EXEC ASM
//ASM.SYSIN DD *
LML       BEGIN
         LM    1,2,0(1)       LOAD ADDRESSES OF X AND Y
         L     5,0(1)         LOAD X INTO R5
         L     7,MASK         LOAD MASK INTO R7
         NR    5,7            SET FRACTIONAL PART OF X TO ZERO
         L     7,EX           LOAD EX INTO R7
         XR    5,7            COMPLIMENT CHARACTERISTIC SETTING SIGN
*                             TO ZERO ,IN R5
         TM    1(1),X'80'     TEST FRACTIONAL PART
         BO    ONE            GO TO 'ONE' IF GREATER THAN 0.5
         TM    1(1),X'40'
         BO    TWO            GO TO 'TWO' IF GREATER THAN 0.25
         TM    1(1),X'20'
         BO    THREE          GO TO 'THREE' IF GREATER THAN 0.125
*                             GOES HERE IF LESS THAN 0.125
         A     5,CTWO         FORM 2**-E IN R5
         B     STORE
THREE    A     5,CONETH
         B     STORE
TWO      A     5,CONETW
         B     STORE
ONE      A     5,CONEON
STORE    ST    5,0(2)         PLACE 2**-E IN STORAGE LOCATION ASSIGNED
*                             TO Y
         SR    8,8
         ST    8,4(2)
         LEAVE
         CNOP  0,8
MASK     DC    X'FF000000'
EX       DC    X'7F000000'
CTWO     DC    X'02800000'
CONETH   DC    X'02400000'
CONETW   DC    X'02200000'
CONEON   DC    X'02100000'
         END
/*
// EXEC ASM
//ASM.SYSIN DD *
GEN      BEGIN
         LM    1,2,0(1)       LOAD ADDRESSES INTO R1 AND R2
         LR    10,2   SAVE LAST RANDOM NUMBER - NEW IX
         LR    9,1
         L     2,0(2)         LOAD INITIAL RANDOM NUMBER INTO R2
         ST    2,IN
         CALL  RN3INZ,(IN)    ENTER INITIAL RANDOM NUMBER INTO RAN3Z
         L     6,END          LOAD 8192 INTO R6
         L     7,COMP         LOAD 'AND' MASK INTO R7
         L     8,SET          LOAD 'OR' MARK INTO R8
         SR    5,5            SET R5 TO ZERO
BGD      CALL  IRN3Z          GENERATE FIRST HALF OF RANDOM NUMBER
         NR    0,7            SET FIRST 8 BITS TO ZEROS
         OR    0,8            SET FIRST 8 BITS TO 01000000...
         ST    0,0(9,5)       STORE FIRST HALF OF RANDOM NUMBER
         CALL  IRN3Z          GENERATE SECOND HALF OF RANDOM NUMBER
         ST    0,0(10)    SAVE LAST RANDOM NUMBER - NEW IX
```

```
         SLL   0,1           SHIFT SECOND HALF 1 BIT LEFT
         ST    0,4(9,5)      STORE SECOND HALF OF RANDOM NUMBER
         A     5,DEC         ADD 8 TO R5
         CLR   5,6           COMPARE R5 TO R6
         BNE   BGD           GO TO BGD IF R5 NOT EQUAL R6
         LEAVE
END      DC    F'8192'
DEC      DC    F'8'
COMP     DC    X'00FFFFFF'
SET      DC    X'40000000'
IN       DS    1F
         END
/*
// EXEC FORTLKGO,TIME.GO=(40,0)
//FORT.SYSIN DD *
C     FORTRAN PARTIAL SIMULATION CODE
C     OPERAND VALUE IS X
      IMPLICIT REAL*8(A-H,L,O-Z,$)
      DIMENSION TWO(100),TTWO(100),LNPLUS(60),LNMINS(60),ARCTAN(60)
      DIMENSION RAND(1024)
      DIMENSION NMPLUS(20),NMZERO(20),NMMINS(20)
      DIMENSION NDPLUS(20),NDZERO(20),NDMINS(20)
      DIMENSION NEPLUS(20),NEZERO(20),NEMINS(20)
      DIMENSION NSPLUS(20),NSZERO(20),NSMINS(20)
      DIMENSION NTPLUS(20),NTZERO(20),NTMINS(20)
      DIMENSION NAPLUS(20),NAZERO(20),NAMINS(20)
      DIMENSION NCPLUS(20),NCZERO(20),NCMINS(20)
      DIMENSION NQPLUS(20),NQZERO(20),NQMINS(20)
C     LISTING OF FORMATS
   11 FORMAT(Z16)
   12 FORMAT(I5,D28.16)
   14 FORMAT(D28.16)
   15 FORMAT(' THE VALUES NPLUS,NZERO,NMINUS ')
   16 FORMAT(3I15)
   18 FORMAT(2D28.16)
   20 FORMAT(I15)
   21 FORMAT(' THE VALUE OF IX TO USE NEXT ')
   24 FORMAT(' MULTIPLICATION:   PLUS,ZERO,MINUS ')
   25 FORMAT(' DIVISION/LOGARITHM:  PLUS,ZERO,MINUS ')
   26 FORMAT(' EXPONENTIAL:  PLUS,ZERO,MINUS ')
   27 FORMAT(' SQUARE ROOT:  PLUS,ZERO,MINUS ')
   28 FORMAT(' TANGENT/COTANGENT:  PLUS,ZERO,MINUS ')
   29 FORMAT (' ARCTANGENT:  PLUS,ZERO,MINUS ')
   30 FORMAT (' COSINE/SINE:  PLUS,ZERO,MINUS ')
   36 FORMAT(4I15)
   38 FORMAT(3D28.16)
   40 FORMAT(4D28.16)
   41 FORMAT ('    I      TWO(I) ')
   42 FORMAT ('    I      TTWO(I) ')
   43 FORMAT ('    I      LNPLUS(I) ')
   44 FORMAT ('    I      LNMINS(I) ')
   45 FORMAT ('    I      ARCTAN(I) ')
   46 FORMAT ('       LN2      ')
   47 FORMAT ('       PI      ')
   48 FORMAT ('       PI2      ')
   49 FORMAT ('       PI4      ')
   51 FORMAT ('       PI8      ')
```

```
   69 FORMAT(' THE PROBABILITY OF A ZERO ')
   70 FORMAT('   MULTIPLICATION   ')
   71 FORMAT ('   DIVISION   ')
   72 FORMAT ('   EXPONENTIAL   ')
   73 FORMAT ('   FIRST SQUARE ROOT   ')
   74 FORMAT ('   TANGENT/COTANGENT   ')
   75 FORMAT ('   ARCTANGENT   ')
   76 FORMAT ('   COSINE/SINE   ')
   77 FORMAT ('   SECOND SQUARE ROOT   ')
C     READ CONSTANTS
      READ(5,11) (TWO(I),I=1,60)
      READ(5,11) (TTWO(I),I=1,60)
      READ(5,11) (LNPLUS(I),I=1,50)
      READ(5,11) (LNMINS(I),I=1,50)
      READ(5,11) (ARCTAN(I),I=1,50)
      READ(5,11) LN2
      READ(5,11) PI
      READ(5,11) PI2
      READ(5,11) PI4
      READ(5,11) PI8
      WRITE (6,41)
      DO 60 I = 1,60
   60 WRITE (6,12) I,TWO(I)
      WRITE (6,42)
      DO 61 I = 1,60
   61 WRITE (6,12) I,TTWO(I)
      WRITE (6,43)
      DO 62 I = 1,50
   62 WRITE (6,12) I,LNPLUS(I)
      WRITE (6,44)
      DO 63 I = 1,50
   63 WRITE (6,12) I,LNMINS(I)
      WRITE (6,45)
      DO 64 I = 1,50
   64 WRITE (6,12) I,ARCTAN(I)
      WRITE (6,46)
      WRITE (6,14) LN2
      WRITE (6,47)
      WRITE (6,14) PI
      WRITE (6,48)
      WRITE (6,14) PI2
      WRITE (6,49)
      WRITE (6,14) PI4
      WRITE (6,51)
      WRITE (6,14) PI8
      DO 100 J = 1,16
      NMPLUS(J) = 0
      NMZERO(J) = 0
      NMMINS(J) = 0
      NDPLUS(J) = 0
      NDZERO(J) = 0
      NDMINS(J) = 0
      NEPLUS(J) = 0
      NEZERO(J) = 0
      NEMINS(J) = 0
      NSPLUS(J) = 0
      NSZERO(J) = 0
```

```
      NSMINS(J) = 0
      NTPLUS(J) = 0
      NTZERO(J) = 0
      NTMINS(J) = 0
      NAPLUS(J) = 0
      NAZERO(J) = 0
      NAMINS(J) = 0
      NCPLUS(J) = 0
      NCZERO(J) = 0
      NCMINS(J) = 0
      NQPLUS(J) = 0
      NQZERO(J) = 0
      NQMINS(J) = 0
  100 CONTINUE
      IX = 78671353
      DO 99200 NNNN = 1,16
      DO 99999 MX = 1,2
      DO 99000 MMM = 1,8
      WRITE(6,21)
      WRITE(6,20) IX
      Z = RAND(KKK)
      IF(Z.GE.0.500) GO TO 300
      IF(Z.GE.0.25) GO TO 250
      IF(Z.LT.0.0625) GO TO 201
      IF(Z.LT.0.125) GO TO 202
      IF(Z.LT.0.1875) GO TO 203
      IF(Z.LT.0.25) GO TO 204
  201 J = 1
      GO TO 500
  202 J = 2
      GO TO 500
  203 J = 3
      GO TO 500
  204 J = 4
      GO TO 500
  250 IF(Z.LT.0.3125) GO TO 255
      IF(Z.LT.0.375) GO TO 256
      IF(Z.LT.0.4375) GO TO 257
      IF(Z.LT.0.500) GO TO 258
  255 J = 5
      GO TO 500
  256 J = 6
      GO TO 500
  257 J = 7
      GO TO 500
  258 J = 8
      GO TO 500
  300 IF(Z.GE.0.750) GO TO 350
      IF(Z.LT.0.5625) GO TO 309
      IF(Z.LT.0.625) GO TO 310
      IF(Z.LT.0.6875) GO TO 311
      IF(Z.LT.0.75) GO TO 312
  309 J = 9
      GO TO 500
  310 J = 10
      GO TO 500
  311 J = 11
```

```
      GO TO 500
  312 J = 12
      GO TO 500
  350 IF(Z.LT.0.8175) GO TO 363
      IF(Z.LT.0.875) GO TO 364
      IF(Z.LT.0.9375) GO TO 365
      IF(Z.LE.1.000) GO TO 366
  363 J = 13
      GO TO 500
  364 J = 14
      GO TO 500
  365 J = 15
      GO TO 500
  366 J = 16
  500 CONTINUE
C     RANDOM NUMBER GENERATOR
      CALL GEN(RAND,IX)
C     MULTIPLICATION ALGORITHM
      X = 0.5*RAND(KKK) + 0.5
      NMINUS = 0
      NZERO = 0
      NPLUS = 0
C     INITIALIZATION
      IF (X.LT.0.75) GO TO 1150
      X = X - 1.0
      GO TO 1160
 1150 X = X - 0.5
 1160 CONTINUE
      NZERO = NZERO + 1
      DO 1400 I = 1,40
      CCM = - TTWO(I+2)
      IF (X.GE.CCM) GO TO 1200
      X = X + TWO(I)
      NPLUS = NPLUS + 1
      GO TO 1400
 1200 CCP = +TTWO(I+2)
      IF (X.GE.CCP) GO TO 1300
      NZERO = NZERO + 1
      GO TO 1400
 1300 X = X - TWO(I)
      NMINUS = NMINUS + 1
 1400 CONTINUE
      NMPLUS(J) = NMPLUS(J) + NPLUS
      NMZERO(J) = NMZERO(J) + NZERO
      NMMINS(J) = NMMINS(J) + NMINUS
C     DIVISION ALGORITHM
      X = 0.5 * RAND(KKK) + 0.5
      NMINUS = 0
      NZERO = 0
      NPLUS = 0
C     INITIALIZATION
      IF(X.LT.0.75) GO TO 1450
      GO TO 1460
 1450 X = X * 2.0D0
 1460 NZERO = NZERO + 1
      DO 2000 I = 1,40
      CCM = 1.0 - TTWO(I+2)
```

```
      IF(X.GT.CCM) GO TO 1500
      X = X * (1.0 + TWO(I))
      NPLUS = NPLUS + 1
      GO TO 2000
 1500 CCP = 1.0 + TTWO(I+2)
      IF(X.GT.CCP) GO TO 1700
      NZERO = NZERO + 1
      GO TO 2000
 1700 X = X * (1.0 - TWO(I))
      NMINUS = NMINUS + 1
 2000 CONTINUE
      NDPLUS(J) = NDPLUS(J) + NPLUS
      NDZERO(J) = NDZERO(J) + NZERO
      NDMINS(J) = NDMINS(J) + NMINUS
C     EXPONENTIAL ALGORITHM
      X = (2.0 * RAND(KKK) - 1.0) * LN2
      NMINUS = 0
      NZERO = 0
      NPLUS = 0
C     INITIALIZATION
      IF(X.LT.0.5) GO TO 3100
      X = X - 0.5
      NPLUS = NPLUS + 1
      GO TO 3800
 3100 IF(X.LT.0.25) GO TO 3200
      X = X - 0.25
      NPLUS = NPLUS + 1
      GO TO 3800
 3200 IF(X.LT.-0.25) GO TO 3300
      NZERO = NZERO + 1
      GO TO 3800
 3300 IF(X.LT.-0.5) GO TO 3400
      X = X + 0.25
      NMINUS = NMINUS + 1
      GO TO 3800
 3400 X = X + 0.5
      NMINUS = NMINUS + 1
 3800 CONTINUE
      DO 4000 I = 1,40
      CCM = - TTWO(I+2)
      IF(X.GT.CCM) GO TO 3850
      X = X - LNMINS(I)
      NMINUS = NMINUS + 1
      GO TO 4000
 3850 CCP = + TTWO(I+2)
      IF(X.GT.CCP) GO TO 3860
      NZERO = NZERO + 1
      GO TO 4000
 3860 X = X - LNPLUS(I)
      NPLUS = NPLUS + 1
 4000 CONTINUE
      NEPLUS(J) = NEPLUS(J) + NPLUS
      NEZERO(J) = NEZERO(J) + NZERO
      NEMINS(J) = NEMINS(J) + NMINUS
C     SQUARE ROOT ALGORITHM: RADIX 2,SCALED
      X = 0.75 * RAND(KKK) + 0.25
      X0 = X
```

```
      NMINUS = 0
      NZERO = 0
      NPLUS = 0
C     TEST INITIAL RANGE OF X
      IF(X.GT.0.3125) GO TO 4100
      CO = 0.500
      R = CO
      RSQ = R * R
      X = XO - RSQ
      NPLUS = NPLUS + 1
      DO 4050 I = 1,40
      CCM = - TTWO(I+3)
      IF(X.GT.CCM) GO TO 4010
      R = R - TWO(I+1)
      RSQ = R * R
      X = XO - RSQ
      NMINUS = NMINUS + 1
      GO TO 4050
 4010 CCP = + TTWO(I+3)
      IF(X.GE.CCP) GO TO 4020
      NZERO = NZERO + 1
      GO TO 4050
 4020 R = R + TWO(I+1)
      RSQ = R * R
      X = XO - RSQ
      NPLUS = NPLUS + 1
 4050 CONTINUE
      GO TO 5000
 4100 IF(X.GT.0.375) GO TO 4200
      CO = 0.5625
      R = CO
      RSQ = R * R
      X = XO - RSQ
      NPLUS = NPLUS + 1
      DO 4150 I = 1,40
      CCM = - TWO(I+2) - TTWO(I+4)
      IF(X.GT.CCM) GO TO 4110
      R = R - TWO(I+1)
      RSQ = R * R
      X = XO - RSQ
      NMINUS = NMINUS + 1
      GO TO 4150
 4110 CCP = - CCM
      IF(X.GE.CCP) GO TO 4120
      NZERO = NZERO + 1
      GO TO 4150
 4120 R = R + TWO(I+1)
      RSQ = R * R
      X = XO - RSQ
      NPLUS = NPLUS + 1
 4150 CONTINUE
      GO TO 5000
 4200 IF(X.GT.0.5625) GO TO 4300
      CO = 0.625
      R = CO
      RSQ = R * R
      X = XO - RSQ
```

```
      NPLUS = NPLUS + 1
      DO 4250 I = 1,40
      CCM = - TWO(I+1)
      IF(X.GT.CCM) GO TO 4210
      R = R - TWO(I+1)
      RSQ = R * R
      X = XO - RSQ
      NMINUS = NMINUS + 1
      GO TO 4250
 4210 CCP = - CCM
      IF(X.GE.CCP) GO TO 4220
      NZERO = NZERO + 1
      GO TO 4250
 4220 R = R + TWO(I+1)
      RSQ = R * R
      X = XO - RSQ
      NPLUS = NPLUS + 1
 4250 CONTINUE
      GO TO 5000
 4300 IF(X.GT.0.875) GO TO 4400
      CO = 0.75
      R = CO
      RSQ = R * R
      X = XO - RSQ
      NPLUS = NPLUS + 1
      DO 4350 I = 1,40
      CCM = - TWO(I+1) - TWO(I+3)
      IF(X.GT.CCM) GO TO 4310
      R = R - TWO(I+1)
      RSQ = R * R
      X = XO - RSQ
      NMINUS = NMINUS + 1
      GO TO 4350
 4310 CCP = - CCM
      IF(X.GE.CCP) GO TO 4320
      NZERO = NZERO + 1
      GO TO 4350
 4320 R = R + TWO(I+1)
      RSQ = R * R
      X = XO - RSQ
      NPLUS = NPLUS + 1
 4350 CONTINUE
      GO TO 5000
 4400 CO = 1.0
      R = CO
      RSQ = R * R
      X = XO - RSQ
      NPLUS = NPLUS + 1
      DO 4450 I = 1,40
      CCM = - TTWO(I+2)
      IF(X.GT.CCM) GO TO 4410
      R = R - TWO(I+1)
      RSQ = R * R
      X = XO - RSQ
      NMINUS = NMINUS + 1
      GO TO 4450
 4410 CCP = - CCM
```

```
      IF(X.GE.CCP) GO TO 4420
      NZERO = NZERO + 1
      GO TO 4450
 4420 R = R + TWO(I+1)
      RSQ = R * R
      X = XO - RSQ
      NPLUS = NPLUS + 1
 4450 CONTINUE
 5000 CONTINUE
      NSPLUS(J) = NSPLUS(J) + NPLUS
      NSZERO(J) = NSZERO(J) + NZERO
      NSMINS(J) = NSMINS(J) + NMINUS
C     TANGENT ALGORITHM
      X = RAND(KKK) * PI2
      R1 = 0.96887271238293440-04
      XH = X
C     TEST RANGE OF OPERAND
      IF(X.LE.R1) GO TO 5100
      IF(X.LE.PI4) GO TO 5200
      C = PI2 - TWO(9)
      IF(X.LT.C) GO TO 5300
      D = PI2 - TWO(40)
      IF(X.LT.D) GO TO 5400
      INDIC = 5
      TANXH = 1.0/(PI2 - XH)
      GO TO 8100
 5100 INDIC = 1
      TANXH = XH
      GO TO 8100
 5200 INDIC = 2
      X = XH
      GO TO 6100
 5300 INDIC = 3
      X = PI2 - XH
      GO TO 6100
 5400 INDIC = 4
      X = PI2 - XH
 6100 NPLUS = 0
      NZERO = 0
      NMINUS = 0
C     INITIALIZATION
      A = 1.0
      B = 0.414213562373095
      X = X - PI8
      NPLUS = NPLUS + 1
C     BEGIN NORMAL ALGORITHM ITERATION
      IF(INDIC.EQ.4) GO TO 6110
      K = 40
      GO TO 6120
 6110 K = 50
 6120 CONTINUE
      DO 7100 I = 1,K
      CCM = - TTWO(I+2)
      IF(X.GE.CCM) GO TO 6200
      X = X + ARCTAN(I)
      SAVEA = A
      A = A + B * TWO(I)
```

```
      B = B - SAVEA * TWO(I)
      NMINUS = NMINUS + 1
      GO TO 6600
 6200 CCP = - CCM
      IF(X.GT.CCP) GO TO 6300
      NZERO = NZERO + 1
      GO TO 6600
 6300 X = X - ARCTAN(I)
      SAVEA = A
      A = A - B * TWO(I)
      B = B + SAVEA * TWO(I)
      NPLUS = NPLUS + 1
 6600 CONTINUE
 7100 CONTINUE
      NTPLUS(J) = NTPLUS(J) + NPLUS
      NTZERO(J) = NTZERO(J) + NZERO
      NTMINS(J) = NTMINS(J) + NMINUS
      IF(INDIC.EQ.2) TANXH = B/A
      IF(INDIC.EQ.3) TANXH = A/B
      IF(INDIC.EQ.4) TANXH = A/B
 8100 CONTINUE
C     ARCTANGENT ALGORITHM
C     TAKE RESULT OF TANGENT ALGORITHM
      X = TANXH
      NMINUS = 0
      NZERO = 0
      NPLUS = 0
C     TEST FOR SPECIAL CASES TO OMIT ALGORITHM
      IF(X.LT.TWO(20)) GO TO 8800
      T = 1.0/TWO(40)
      IF(X.GT.T) GO TO 8800
      D = DABS(X-1.0)
      IF(D.LT.TWO(19)) GO TO 8800
      IF(X.GE.1.0) GO TO 8601
C     HERE X IS LESS THAN UNITY
      A = X + 1.0
      B = X - 1.0
      IF(X.GT.0.25) GO TO 8300
      A = 1.0
      B = X
      NPLUS = NPLUS + 1
      GO TO 8510
 8300 IF(X.GT.0.5) GO TO 8400
      SAVEA = A
      A = 0.5 * A - 0.25 * B
      B = 0.5 * B + 0.25 * SAVEA
      NPLUS = NPLUS + 1
      GO TO 8510
 8400 A = A * 0.5
      B = B * 0.5
      NZERO = NZERO + 1
 8510 CONTINUE
C     BEGIN NORMAL ALGORITHM ITERATION WITH R = 40
      DO 8600 I = 1,40
      CCM = - TTWO(I+2)
      IF(B.LE.CCM) GO TO 8520
      CCP = - CCM
```

```
      IF(B.GT.CCP) GO TO 8530
      NZERO = NZERO + 1
      GO TO 8540
 8520 SAVEA = A
      A = A - B * TWO(I)
      B = B + SAVEA * TWO(I)
      NMINUS = NMINUS + 1
      GO TO 8540
 8530 SAVEA = A
      A = A + B * TWO(I)
      B = B - SAVEA * TWO(I)
      NPLUS = NPLUS + 1
 8540 CONTINUE
      IF(I.EQ.40) GO TO 8700
 8600 CONTINUE
 8601 CONTINUE
C     HERE X IS AT LEAST UNITY
C     FIND EXPONENT Y OF OPERAND X
      CALL LML(X,Y)
C     LML RETURNS Y = 2**(-E)
      P = Y * X
      A = P + Y
      B = P - Y
      IF (X.LT.1.5) GO TO 8605
      SAVEA = A
      A = A + B
      B = B - SAVEA
      NMINUS = NMINUS + 1
 8605 NZERO = NZERO + 1
        IF(A.GT.1.25) GO TO 8610
      A = A * 0.75
      B = B * 0.75
      NPLUS = NPLUS + 1
      GO TO 8670
 8610 IF(A.GT.1.5) GO TO 8620
      A = A * 0.625
      B = B * 0.625
      NPLUS = NPLUS + 1
      GO TO 8670
 8620 A = A * 0.5
      B = B * 0.5
      NZERO = NZERO + 1
 8670 CONTINUE
C     BEGIN NORMAL ALGORITHM ITERATION WITH R = 40
      DO 8700 I = 1,40
      CCM = - TTWO(I+2)
      IF(B.LE.CCM) GO TO 8680
      CCP = - CCM
      IF(B.GT.CCP) GO TO 8690
      NZERO = NZERO + 1
      GO TO 8700
 8680 SAVEA = A
      A = A - B * TWO(I)
      B = B + SAVEA * TWO(I)
      NMINUS = NMINUS + 1
      GO TO 8700
 8690 SAVEA = A
```

```
      A = A + B * TWO(I)
      B = B - SAVEA * TWO(I)
      NPLUS = NPLUS + 1
 8700 CONTINUE
      NAPLUS(J) = NAPLUS(J) + NPLUS
      NAZERO(J) = NAZERO(J) + NZERO
      NAMINS(J) = NAMINS(J) + NMINUS
      GO TO 8900
 8800 CONTINUE
 8900 CONTINUE
C     COSINE/SINE ALGORITHM
      X = RAND(KKK) * PI4
      NMINUS = 0
      NZERO = 0
      NPLUS = 0
C     INITIALIZATION
      X = X - PI8
      NPLUS = NPLUS + 1
      NZERO = NZERO + 1
C     FIRST QUARTER NON-REDUNDANT
      DO 24000 I = 2,9
      IF(X.GT.0) GO TO 23000
      X = X + ARCTAN(I)
      NPLUS = NPLUS + 1
      GO TO 24000
23000 X = X - ARCTAN(I)
      NMINUS = NMINUS + 1
24000 CONTINUE
C     REST REDUNDANT
      DO 29000 I = 10,40
      CCP = TTWO(I+2)
      IF(X.GT.CCP) GO TO 26000
      CCM = - CCP
      IF(X.LE.CCM) GO TO 27000
      NZERO = NZERO + 1
      GO TO 29000
26000 X = X - ARCTAN(I)
      IF(I.LE.20) NPLUS = NPLUS + 1
26100 NPLUS = NPLUS + 1
      GO TO 29000
27000 X = X + ARCTAN(I)
      IF(I.LE.20) NMINUS = NMINUS + 1
27100 NMINUS = NMINUS + 1
29000 CONTINUE
      NCPLUS(J) = NCPLUS(J) + NPLUS
      NCZERO(J) = NCZERO(J) + NZERO
      NCMINS(J) = NCMINS(J) + NMINUS
C     SQUARE ROOT ALGORITHM. HIGHER RADIX, NOT SCALED
      DO 30000 I = 61,100
      TWO(I) = 0.0
30000 TTWO(I) = 0.0
      X = 0.75 * RAND(KKK) + 0.25
      Y = X
      NMINUS = 0
      NZERO = 0
      NPLUS = 0
      ALPHA = X
```

```
C     INITIAL STEP
      IF(X.GE.0.5) GO TO 30100
      X = X * 4.0
      ALPHA = ALPHA * 2.0
30100 NZERO = NZERO + 1
      DO 39000 I = 1,40
      CCP = 1.0 + TTWO(I+2)
      IF(X.GE.CCP) GO TO 30200
      CCM = 1.0 - TTWO(I+2)
      IF(X.GT.CCM) GO TO 30300
      X = X * (1.0 + TWO(I) + TWO(2*I+2))
      ALPHA = ALPHA * (1.0 + TWO(I+1))
      NPLUS = NPLUS + 1
      GO TO 30400
30200 X = X * (1.0 - TWO(I) + TWO(2*I+2))
      ALPHA = ALPHA * (1.0 - TWO(I+1))
      NMINUS = NMINUS + 1
      GO TO 30400
30300 NZERO = NZERO + 1
30400 CONTINUE
39000 CONTINUE
      NQPLUS(J) = NQPLUS(J) + NPLUS
      NQZERO(J) = NQZERO(J) + NZERO
      NQMINS(J) = NQMINS(J) + NMINUS
99000 CONTINUE
      WRITE(6,24)
      WRITE (6,36) (I,NMPLUS(I),NMZERO(I),NMMINS(I),I=1,16)
      WRITE(6,25)
      WRITE(6,36) (I,NDPLUS(I),NDZERO(I),NDMINS(I),I=1,16)
      WRITE(6,26)
      WRITE(6,36) (I,NEPLUS(I),NEZERO(I),NEMINS(I),I=1,16)
      WRITE(6,27)
      WRITE(6,36) (I,NSPLUS(I),NSZERO(I),NSMINS(I),I=1,16)
      WRITE(6,28)
      WRITE(6,36) (I,NTPLUS(I),NTZERO(I),NTMINS(I),I=1,16)
      WRITE(6,29)
      WRITE(6,36) (I,NAPLUS(I),NAZERO(I),NAMINS(I),I=1,16)
      WRITE(6,30)
      WRITE(6,36) (I,NCPLUS(I),NCZERO(I),NCMINS(I),I=1,16)
      WRITE(6,27)
      WRITE(6,36) (I,NQPLUS(I),NQZERO(I),NQMINS(I),I=1,16)
99999 CONTINUE
C     CALCULATE PROBABILITY OF A ZERO FROM THESE DATA
      MMULT = 0
      MMULTZ = 0
      MDIV = 0
      MDIVZ = 0
      MEXP = 0
      MEXPZ = 0
      MSQT1 = 0
      MSQT1Z = 0
      MTAN = 0
      MTANZ = 0
      MATN = 0
      MATNZ = 0
      MCOS = 0
      MCOSZ = 0
```

```
      MSQT2 = 0
      MSQT2Z = 0
      DO 99100 I = 1,16
      MMULT = MMULT + NMPLUS(I) + NMMINS(I)
      MMULTZ = MMULTZ + NMZERO(I)
      MDIV = MDIV + NDPLUS(I) + NDMINS(I)
      MDIVZ = MDIVZ + NDZERO(I)
      MEXP = MEXP + NEPLUS(I) + NEMINS(I)
      MEXPZ = MEXPZ + NEZERO(I)
      MSQT1 = MSQT1 + NSPLUS(I) + NSMINS(I)
      MSQT1Z = MSQT1Z + NSZERO(I)
      MTAN = MTAN + NTPLUS(I) + NTMINS(I)
      MTANZ = MTANZ + NTZERO(I)
      MATN = MATN + NAPLUS(I) + NAMINS(I)
      MATNZ = MATNZ + NAZERO(I)
      MCOS = MCOS + NCPLUS(I) + NCMINS(I)
      MCOSZ = MCOSZ + NCZERO(I)
      MSQT2 = MSQT2 + NQPLUS(I) + NQMINS(I)
      MSQT2Z = MSQT2Z + NQZERO(I)
99100 CONTINUE
      AMULT = MMULT + MMULTZ
      AMULT = NMULTZ
      PMULT = AMULTZ/AMULT
      WRITE (6,78)
      WRITE (6,14) PMULT
      DIV = MDIV + MDIVZ
      DIVZ = MDIVZ
      PDIV = DIVZ/DIV
      EXP = MEXP + MEXPZ
      EXPZ = MEXPZ
      PEXP = EXPZ/EXP
      SQT1 = MSQT1 + MSQT1Z
      SQT1Z = MSQT1Z
      PSQT1 = SQT1Z/SQT1
      TAN = MTAN + MTANZ
      TANZ = MTANZ
      PTAN = TANZ/TAN
      ATN = MATN + MATNZ
      ATNZ = MATNZ
      PATN = ATNZ/ATN
      COS = MCOS + MCOSZ
      COSZ = MCOSZ
      PCOS = COSZ/COS
      SQT2 = MSQT2 + MSQT2Z
      SQT2Z = MSQT2Z
      PSQT2 = SQT2Z/SQT2
      WRITE(6,69)
      WRITE (6,71)
      WRITE(6,14) PMULT
      WRITE (6,14) PDIV
      WRITE (6,72)
      WRITE (6,14) PEXP
      WRITE (6,73)
      WRITE (6,14) PSQT1
      WRITE (6,74)
      WRITE (6,14) PTAN
      WRITE (6,75)
```

```
      WRITE (6,14) PATN
      WRITE (6,76)
      WRITE (6,14) PCOS
      WRITE (6,77)
      WRITE (6,14) PSQT2
99200 CONTINUE
      WRITE(6,21)
      WRITE(6,20) IX
C     CONSTANTS IN IBM 360 HEXADECIMAL FORMAT
      STOP
      END
//GO.SYSIN DD *
$ENTRY
4080000000000000                                                        1
4040000000000000                                                        2
4020000000000000                                                        3
4010000000000000                                                        4
3F80000000000000                                                        5
3F40000000000000                                                        6
3F20000000000000                                                        7
3F10000000000000                                                        8
3E80000000000000                                                        9
3E40000000000000                                                       10
3E20000000000000                                                       11
3F10000000000000                                                       12
3D80000000000000                                                       13
3D40000000000000                                                       14
3D20000000000000                                                       15
3D10000000000000                                                       16
3C80000000000000                                                       17
3C40000000000000                                                       18
3C20000000000000                                                       19
3C10000000000000                                                       20
3B80000000000000                                                       21
3B40000000000000                                                       22
3B20000000000000                                                       23
3B10000000000000                                                       24
3A80000000000000                                                       25
3A40000000000000                                                       26
3A20000000000000                                                       27
3A10000000000000                                                       28
3980000000000000                                                       29
3940000000000000                                                       30
3920000000000000                                                       31
3910000000000000                                                       32
3880000000000000                                                       33
3840000000000000                                                       34
3820000000000000                                                       35
3810000000000000                                                       36
3780000000000000                                                       37
3740000000000000                                                       38
3720000000000000                                                       39
3710000000000000                                                       40
3680000000000000                                                       41
3640000000000000                                                       42
3620000000000000                                                       43
3610000000000000                                                       44
```

3580000000000000	45
3540000000000000	46
3520000000000000	47
3510000000000000	48
3480000000000000	49
3440000000000000	50
3420000000000000	51
3410000000000000	52
3380000000000000	53
3340000000000000	54
3320000000000000	55
3310000000000000	56
3280000000000000	57
3240000000000000	58
3220000000000000	59
3210000000000000	60
4118000000000000	1
40C0000000000000	2
4060000000000000	3
4030000000000000	4
4018000000000000	5
3FC0000000000000	6
3F60000000000000	7
3F30000000000000	8
3F18000000000000	9
3EC0000000000000	10
3E60000000000000	11
3E30000000000000	12
3E18000000000000	13
3DC0000000000000	14
3D60000000000000	15
3D30000000000000	16
3D18000000000000	17
3CC0000000000000	18
3C60000000000000	19
3C30000000000000	20
3C18000000000000	21
3BC0000000000000	22
3B60000000000000	23
3B30000000000000	24
3B18000000000000	25
3AC0000000000000	26
3A60000000000000	27
3A30000000000000	28
3A18000000000000	29
39C0000000000000	30
3960000000000000	31
3930000000000000	32
3918000000000000	33
38C0000000000000	34
3860000000000000	35
3830000000000000	36
3818000000000000	37
37C0000000000000	38
3760000000000000	39
3730000000000000	40
3718000000000000	41

36C0000000000000	42
3660000000000000	43
3630000000000000	44
3618000000000000	45
35C0000000000000	46
3560000000000000	47
3530000000000000	48
3518000000000000	49
34C0000000000000	50
3460000000000000	51
3430000000000000	52
3418000000000000	53
33C0000000000000	54
3360000000000000	55
3330000000000000	56
3318000000000000	57
32C0000000000000	58
3260000000000000	59
3230000000000000	60
4067CC8FB2FE6126	1
40391FEF8F35343C	2
401E27076E2AF2E4	3
3FF85186008B152C	4
3F7E0A6C39E0CBF2	5
3F3F815161F807C6	6
3F1FE02A6B106784	7
3EFF805515885DFE	8
3E7FE00AA6AC4398	9
3E3FF80155156150	10
3E1FFE002AA6AB0A	11
3DFFF80055515582	12
3D7FFE000AAA6AAA	13
3D3FFF8001555154	14
3D1FFFE0002AAA6A	15
3CFFFF8000555514	16
3C7FFFE00007FFFE	17
3C3FFFF80000FFFE	18
3C1FFFFE00001FFF	19
3BFFFFF800003FFE	20
3B7FFFFE000007FE	21
3B3FFFFF800000FE	22
3B1FFFFFE000001F	23
3AFFFFFF8000003E	24
3A7FFFFFE0000006	25
3A40000000000000	26
3A20000000000000	27
3A10000000000000	28
3980000000000000	29
3940000000000000	30
3920000000000000	31
3910000000000000	32
3880000000000000	33
3840000000000000	34
3820000000000000	35
3810000000000000	36
3780000000000000	37
3740000000000000	38

3720000000000000	39
3710000000000000	40
3680000000000000	41
3640000000000000	42
3620000000000000	43
3610000000000000	44
3580000000000000	45
3540000000000000	46
3520000000000000	47
3510000000000000	48
3480000000000000	49
3440000000000000	50
C0B17217F7D1CF62	1
C049A58844D36E40	2
C0222F1D044FC8EE	3
C0108598B59E3A05	4
BF820AEC4F3A2210	5
BF408159624D611C	6
BF20202AEB11BCDC	7
BF10080559588B35	8
BE80200AAEAC44EC	9
BE40080155956156	10
BE2002002AAEAB0A	11
BE10008005559558	12
BD8002000AAAEAAA	13
BD40008001555954	14
BD200020002AAAEA	15
BD10000800055559	16
BC80002000080002	17
BC40000800010000	18
BC20000200002000	19
BC10000080000400	20
BB80000200000800	21
BB40000080000100	22
BB20000020000020	23
BB10000008000004	24
BA80000020000008	25
BA40000000000000	26
BA20000000000000	27
BA10000000000000	28
B980000000000000	29
B940000000000000	30
B920000000000000	31
B910000000000000	32
B880000000000000	33
B840000000000000	34
B820000000000000	35
B810000000000000	36
B780000000000000	37
B740000000000000	38
B720000000000000	39
B710000000000000	40
B680000000000000	41
B640000000000000	42
B620000000000000	43
B610000000000000	44
B580000000000000	45

```
B540000000000000   46
B520000000000000   47
B510000000000000   48
B480000000000000   49
B440000000000000   50
4076B19C1586ECF9    1
403EB6EBF25901A1    2
401FD5BA9AAC2F65    3
3FFFAADDB967EF20    4
3F7FF556EEA5D879    5
3F3FFEAAB776E530    6
3F1FFFD555BBBA94    7
3EFFFFAAAADDDDA2    8
3E7FFFF55556EEEA    9
3E3FFFFEAAAAB774   10
3E1FFFFFD55555B9   11
3DFFFFFFAAAAAAD4   12
3D7FFFFFF5555551   13
3D3FFFFFFEAAAAA7   14
3D1FFFFFFFD55553   15
3CFFFFFFFFAAAA9F   16
3C7FFFFFFFF5554F   17
3C3FFFFFFFFEAAA7   18
3C1FFFFFFFFFD553   19
3BFFFFFFFFFFFAA9F   20
3B80000000000000   21
3B40000000000000   22
3B20000000000000   23
3B10000000000000   24
3A80000000000000   25
3A40000000000000   26
3A20000000000000   27
3A10000000000000   28
3980000000000000   29
3940000000000000   30
3920000000000000   31
3910000000000000   32
3880000000000000   33
3840000000000000   34
3820000000000000   35
3810000000000000   36
3780000000000000   37
3740000000000000   38
3720000000000000   39
3710000000000000   40
3680000000000000   41
3640000000000000   42
3620000000000000   43
3610000000000000   44
3580000000000000   45
3540000000000000   46
3520000000000000   47
3510000000000000   48
3480000000000000   49
3440000000000000   50
40B17217F7D1CF62   66
413243F6A8885A2F   61
```

```
411921FB54442D19                                                        62
40C90FDAA22168C1                                                        63
406487ED5110B45D                                                        64
$STOP
/*
```

BIBLIOGRAPHY

Ashenhurst, R. L., "Function Evaluation in Unnormalized Arithmetic," Journal of the ACM, 11:2:168-187, April, 1964.

Ashenhurst, R. L., N. Metropolis, "Unnormalized Floating Point Arithmetic," Journal of the ACM, 6:3:415-428, July, 1959.

Avizienis, A., "Arithmetic Microsystems for Synthesis of Function Generators," Proceedings of the IEEE, 54:12:1910-1919, December, 1966.

Bemer, R. W., "A Machine Method for Square-Root Computation," Communications of the ACM, 1:1:6-7, January, 1958.

Bemer, R. W., "A Note on Range Transformation for Square Root and Logarithm," Communications of the ACM, 6:6:306-307, June, 1963.

Bemer, R. W., "A Subroutine Method for Calculating Logs," Communications of the ACM, 1:5:5-7, May, 1958.

Cantor, D., G. Estrin, R. Turn, "Logarithmic and Exponential Function Evaluation in a Variable Structure Digital Computer," IRE Transactions on Electronic Computers, EC-11:2:155-164, April, 1962.

Cochran, W. G., Sampling Techniques, New York, John Wiley & Sons, Inc., 1963.

Combet, M., H. Van Zonneveld, L. Verbeek, "Computation of the Base Two Logarithm of Binary Numbers," IEEE Transactions on Electronic Computers," EC-14:6:863-868, December, 1965.

Cowgill, D. "Logic Equations for a Built-In Square Root Method, "IEEE Transactions on Electronic Computers, EC-13:2:156-157, April, 1964.

Frieman, C. V., "Statistical Analysis of Certain Binary Division Algorithms," Proceedings of the IRE, 49:1:91-103, January, 1961.

Garner, L., "Number Systems and Arithmetic," Advances in Computers, Vol. 6, New York, Academic Press Inc., 1965.

Gilman, R. E., "A Mathematical Procedure for Machine Division," Communications of the ACM, 2:4:10-12, April, 1959.

Goto, M., T. Fukumura, "Application of Generalized Minimal Representation of Binary Numbers to Square Rooting," Electronics and Communications in Japan, 50:5:122-129, May, 1967.

Hart, J. F., Handbook of Computer Approximations, New York, John Wiley & Sons, Inc., 1968.

Hastings, C., Approximations for Digital Computers, Princeton, New Jersey, Princeton University Press, 1955.

Kish, L., Survey Sampling, New York, John Wiley & Sons, Inc., 1965.

Kogbetliantz, E. G., "Computation of Arcsine N for N Between 0 and 1 Using an Electronic Computer," IBM Journal of Research and Development, 2:218-222, July, 1958.

Kogbetliantz, E. G., "Computation of Arctan N for N Between Plus and Minus Infinity Using an Electronic Computer," IBM Journal of Research and Development, 2:43-53, January, 1958.

Kogbetliantz, E. G., "Computation of E to the N for N Between Plus and Minus Infinity Using an Electronic Computer," IBM Journal of Research and Development, 1:110-115, April, 1957.

Kogbetliantz, E. G., "Computation of Sin N, Cos N and M^{th} Root of N Using an Electronic Computer," IBM Journal of Research and Development, 3:147-152, April, 1959.

Lenaerts, E. H., "Automatic Square Rooting," Electronic Engineering, 27:287-289, July, 1955.

MacSorley, O. L., "High Speed Arithmetic in Binary Computers," Proceedings of the IRE, 49:1:67-91, January, 1961.

Meggitt, J. E., "Pseudo Division and Pseudo Multiplication Processes," IBM Journal of Research and Development, 6:2:210-226, April, 1962.

Metze, G., "A Class of Binary Divisions Yielding Minimally Represented Quotients," IRE Transactions on Electronic Computers, EC-11:6:761-764, December, 1962.

Metze, G., "Minimal Square Rooting," IEEE Transactions on Electronic Computers, EC-14:2:181-185, April, 1965.

Penhollow, J. O., "A Study of Arithmetic Recoding with Applications in Multiplication and Division," Ph.D. Thesis, University of Illinois, September, 1962.

Reitwiesner, G. W., "Binary Arithmetic," Advances in Computers, Vol. 1, New York, Academic Press Inc., 1960.

Robertson, J. E., "A New Class of Digital Division Methods," IRE Transactions on Electronic Computers, EC-7:5:218-222, September, 1958.

Robertson, J. E., "Increasing the Efficiency of Digital Computer Arithmetic Through Use of Redundancy," Lecture Notes for EE 497 B, University of Illinois, Fall Semester, 1964.

Robertson, J. E., "The Correspondence Between Methods of Digital and Multiplier Recoding Procedures," University of Illinois DCL Report No. 252, December 21, 1967.

Sarafayan, D., "Divisionless Computations of Square Roots Through Continued Squaring," Communications of the ACM, 3:5:319-321, May, 1960.

Schreider, Y. A., Method of Statistical Testing Monte Carlo Method, New York, Elsevier Publishing Co., 1964.

Senzig, D. N., "Digit-by-Digit Generation of the Trigonometric and Hyperbolic Functions," IBM Research Report, RC-860, December 17, 1962.

Shively, R. R., "Stationary Distributions of Partial Remainders in SRT Digital Division," Ph.D. Thesis, University of Illinois, June, 1963.

Specker, W. H., "A Class of Algorithms for Ln X, Exp X, Sin X, Cos X, Tan^{-1} X, and Cot^{-1} X," IEEE Transactions on Electronic Computers, EC-14:1:85-86, February, 1965.

Tung, C., "A Combinational Arithmetic Function Generation System," Ph.D. Thesis, UCLA Report No. 68-29, June, 1968.

Volder, J. E., "The CORDIC Trigonometric Computing Technique," IEEE Transactions on Electronic Computers, EC-8:5:330-334, September, 1959.

VITA

Bruce Gene De Lugish was born on March 16, 1943 in Davenport, Iowa. After graduation from high school in Rock Island, Illinois, he earned the Bachelor of Science, Master of Science, and Doctor of Philosophy degrees, all from the University of Illinois at Urbana, in June, 1965, August, 1968, and June, 1970, respectively.

As an undergraduate he was elected to three honor societies: Eta Kappa Nu, Tau Beta Pi, and Sigma Tau. He was also chosen Outstanding Senior in the Men's Residence Halls Association in 1965. As a graduate he was invited to membership in Phi Kappa Phi.

During graduate work he was a teaching fellow and teaching assistant in the Department of Electrical Engineering and a research assistant in the Department of Computer Science.

Summer employment included work with Bell Telephone Laboratories at Holmdel, New Jersey and with Lawrence Radiation Laboratory at Livermore, California.

Mr. De Lugish is a member of IEEE and ACM.